Introduction to Database Reverse Engineering

Hainaut, Jean-Luc

Publication date:
2002

Introduction to Database Reverse Engineering

Jean-Luc Hainaut

May 2002

LIBD - Laboratory of Database Application Engineering
Institut d'Informatique - University of Namur

rue Grandgagnage, 21 • B-5000 Namur (Belgium)
jlh@info.fundp.ac.be – http://www.info.fundp.ac.be/libd

Credits

This text is based on results of three R&D projects, namely TRAMIS, addressing the baselines of transformational support to database engineering [Hainaut 1992], PHENIX, dedicated to *AI techniques for Database Reverse Engineering* [Joris 1992], and DB-MAIN, coping with *Methods and Tools for Information System Evolution* [Hainaut 1994]. So, thanks to Muriel Chandelon, Catherine Tonneau and Michel Joris (PHENIX), as well as to Mario Cadelli, Bernard Decuyper and Olivier Marchand (TRAMIS).

The current shape and contents of this text is strongly influenced by the work of the DB-MAIN team: Vincent Englebert (Meta tools, Voyager 2), Jean Henrard (reverse engineering, program understanding), Jean-Marc Hick (repository, GUI, DB evolution control), Didier Roland (tool assistants, design process modeling). Besides their theoretical contribution, the rich DB-MAIN CASE environment, of which they are the architects and craftmen, transforms the material of this book into practical methods and techniques that can be most helpful for practitioners. Quite naturally, they must be considered as co-authors of this text.

Summary

This text is an **extended abstract** for a future book devoted to database reverse engineering (DBRE), considered as a specific, but general purpose, activity of information system engineering, and particularly system reengineering. To give it as large a scope as possible, we have developed general principles that can easily be specialized for any actual data management system, ranging from simple file managers (COBOL or RPG) to modern DB managers such as relational systems and OO DBMS. These principles are organized as a general DBRE methodology that is built in a systematic way. This methodology can be specialized according to specific DBMS models. It can be used to compare methodologies proposed in the literature and through reverse engineering CASE tools.

First, the baselines of the realm are developed: what are the problems (1), how to describe and transform technical and semantic data structures (2), and where are the problem coming from (3). Then, the principles of a generic methodology are developed (4). It comprises two main processes, namely Data Structure Extraction (5), through which the complete technical structures are elicited, and Data Structure Conceptualization (6), that tries to interpret these data structures into conceptual terms. The methodology is specialized according to some popular data models (7) and the way in which current CASE technology can support the processes is discussed (8). A short, but realistic, case study of database reeingineering is presented (9). A state of the art (10) and bibliographic notes (11) close the presentation.

Table of contents (tentative)

1. Introduction

2. Data schema specification
- A wide-spectrum specification model
- Schema transformation

3. Database development revisited
- Database design framework
- Transformational description of the Logical design process
- Optimization techniques
- Expressing and coding non-DMS constructs
- Analysis of semantics preservation in empirical database design

4. A general database reverse engineering methodology
- Deriving a transformational DBRE model
- The major phases of the DBRE methodology
- The Data Structure Extraction phase
- The Data Structure Conceptualization phase

5. The data structure extraction process
- Explicit vs implicit constructs
- Explicit constructs extraction
- The Refinement process: information sources and elicitation techniques
- The Refinement process: representative problems
- The Refinement process: application to foreign key elicitation

6. The data structure conceptualization process
- Basic conceptualization
- Schema Untranslation
- Schema De-optimization
- Conceptual normalization

7. DMS-oriented DBRE methodologies
- Standard file DBRE
- Hierarchical database DBRE
- Network database DBRE
- Shallow database DBRE
- Relational database DBRE
- Other standard DBMS DBRE
- Object-oriented database DBRE

8. CASE technology for DBRE
Requirements
Project and document representation and management
Support for the data structure extraction process
Support for the data structure conceptualization process
The DB-MAIN CASE environment
State of the art in database CARE tools

9. A case study: Database Migration
Project preparation
Data structure extraction
Data structure conceptualization
Database conversion
The COBOL source code (excerpts)

10. State of the art and conclusion

11. References

Chapter 1

Introduction

Every information system that has been developed and maintained decently is assumed to have a complete and up to date technical and functional documentation associated with it. Each of its components is described in detail and the way it was developed is clearly motivated. As a consequence, fixing the rare remaining bugs, introducing new functionalities, changing the architecture, porting parts on another platform, interfacing them with foreign systems or extracting components to include in, say, a distributed object architecture all are child play.

So, the term *legacy system* should raise in our mind a feeling of admiration and respect, in consideration to the experienced developers whose skill left us such beautiful artcraft masterpieces.

Unfortunately, not everybody shares this view:

> *A legacy IS is any IS that significantly resists modifications and change. Typically, a legacy IS is big, with millions of lines of code, and more than 10 years old.* [Brodie 1995]

Since no change can be made to an information system before we can get a precise and detailed knowledge on its functional and technical aspects, there is a strong need for scientifically rebuilding the lost documentation of current systems that are to be maintained and to evolve in a world of increasingly complex technology.

Database reverse engineering

Reverse engineering (RE) a piece of software consists, among others, in recovering or reconstructing its functional and technical specifications, starting mainly from the source text of the programs [Chikofski 1990] [IEEE 1990] [Hall 1992].

The problem is particularly complex with old and ill-designed applications. In this case, not only no decent documentation (if any) can be relied on, but the lack of systematic methodologies for designing and maintaining them have led to tricky and obscure code. Therefore, reverse engineering has long been recognized as a complex, painful and prone-to-failure activity, in such a way that it is simply not undertaken most of the time, leaving huge amounts

of invaluable knowledge buried in the programs, and therefore definitively lost.

In information systems, or data-oriented applications, i.e. in applications whose central component is a database or a set of permanent files, the complexity can be broken down by considering that the files or databases can be reverse engineered (almost) independently of the procedural parts. Indeed,

- the semantic distance between the so-called conceptual specifications and the physical implementation is most often narrower for data than for procedural parts (a COBOL file structure is easier to understand than a COBOL procedure);
- the permanent data structures are generally the most stable part of applications;
- even in very old applications, the semantic structures that underlie the file structures are mainly procedure-independent, though their physical structure is highly procedure-dependent;
- reverse engineering the procedural part of an application is much easier when the semantic structure of the data has been elicited.

Therefore, concentrating on reverse engineering the data components of the application first can be much more efficient than trying to cope with the whole application.

Being a complex process, database reverse engineering (DBRE) cannot be successful without the support of adequate tools. An increasing number of commercial products (claim to) offer DBRE functionalities. Though they ignore many of the most difficult aspects of the problem, these tools provide their users with invaluable help to carry out DBRE more effectively [Rock 1990].

Motivation and objectives

Reverse engineering is just one step in the information system life cycle. Indeed, painfully recovering the specifications of a database is not a sufficient motivation *per se*. It is generally intended to redocument, convert, restructure, maintain or extend legacy applications. Here follow some of the most frequent objectives of database reverse engineering.

- *Knowledge acquisition in system development*. During the development of a new system, one of the early phases consists in gathering and formalizing users requirements from various sources such as users interviews and corporate documents analysis. In many cases, some partial implementation of the future system may exist already, for instance in the form of a user-developed small system, the analysis of which can bring early useful information.
- *System maintenance*. Fixing bugs and modifying system functions require understanding the concerned component, including, in data-centered systems, the semantics and the implementation of the permanent data structures.
- *System reengineering*. Reengineering a system is changing its internal architecture or rewriting the code of some components without modifying the external specifications. The overall goal is to restart with a cleaner implementation that should make further maintenance and evolution easier. Quite obviously, the technical aspects as well as its

functional specifications have to be clearly understood. The same will be true for the other three objectives whose description follows.

- *System extension*. This term designates changing and augmenting the functional goals of a system, such as adding new functions, or its external behavior, such as improving its robustness.
- *System migration*. Migrating a system consists in replacing one or several of the implementation technologies. IMS/DB2, COBOL/C, monolithic/Client-server, centralized/distributed are some widespread examples of system migration.
- *System integration*. Integrating two or more systems yields a unique system that includes the functions and the data of the former. The resulting system can be physical, in which case it has been developed as a stand-alone application, or it can be a virtual system in which a dynamic interface translates the global queries into local queries addressed to the source systems. The most popular form of virtual integrated system is the *federated database* architecture.
- *Quality assessment*. Analyzing the code and the data structures of a system in some detail can bring useful hints about the quality of this system, and about the way it was developed. M. Blaha argues that assessing the quality of a vendor software database through DBRE techniques is a good way to evaluate the quality of the whole system [Blaha 1998b].
- *Data extraction/conversion.* In some situations, the only component to salvage when abandoning a legacy system is its database. The data have to be converted into another format, which requires some knowledge on its physical and semantic characteristics. On the other hand, most datawarehouses are filled with aggregated data that are extracted from corporate databases. This transfer requires a deep understanding of the physical data structures, to write the extraction routines, and of their semantics, to interpret them correctly.
- *Data Administration*. DBRE is also required when developing a data administration function that has to know and record the description of all the information resources of the organization.
- *Component reuse*. In emerging system architectures, reverse engineering allows developers to identify, extract and wrap legacy functional and data components in order to integrate them in new systems, generally through ORB technologies [Sneed 1996] [Thiran 1998].

Specific DBRE problems

Experience quickly teaches that recovering conceptual data structures can be much more complex than merely analyzing the DDL[1] code of the database. Untranslated data structures and constraints, non standard implementation approaches and techniques and ill-designed

1. The *Data Description Language* (DDL) is that part of the database management system facilities intended to declare or build the data structures of the database.

schemas are some of the difficulties that the analysts encounter when trying to understand existing databases from operational system components. Since the DDL code no longer is the unique information source, the analyst is forced to refer to other documents and system components that will prove more complex and less reliable. The most frequent sources of problems have been identified [Andersson 1994], [Blaha 1995], [Hainaut 1993b], [Petit 1994], [Premerlani 1993] and can be classified as follows.

- *Lack of documentation*. Every piece of software should be accompanied by an up to date documentation. However, it is hard to hide the fact that most of them live undocumented, or, at best, are supported by poor, partial, heterogeneous, inconsistent and obsolete documentation, generally on paper.
- *Weakness of the DBMS models*. The technical model provided by the DMS[2], such as CODASYL-like systems, standard file managers and IMS DBMS, can express only a small subset of the structures and constraints of the intended conceptual schema. In favorable situations, these discarded constructs are managed in procedural components of the application: programs, dialog procedures, triggers, etc., and can be recovered through procedural analysis.
- *Implicit structures*. Such constructs have intentionally not been explicitly declared in the DDL specification of the database. They have generally been implemented in the same way as the discarded constructs mentioned above.
- *Optimized structures*. For technical reasons, such as time and/or space optimization, many database structures include non semantic constructs. In addition, redundant and unnormalized constructs are added to improve response time.
- *Awkward design*. Not all databases were built by experienced designers. Novice and untrained developers, generally unaware of database theory and database methodology, often produce poor or even wrong structures.
- *Obsolete constructs*. Some parts of a database have been abandoned, and ignored by the current programs.
- *Cross-model influence*. The professional background of designers can lead to very peculiar results. For instance, some relational databases are actually straightforward translations of IMS databases, of COBOL files or of spreadsheets [Blaha 1995].

About this book

This book is an introduction to the problem of database reverse engineering, considered as a specific, but general purpose, activity of information system engineering, and particularly system reengineering. To give it as large a scope as possible, we have developed general principles that can easily be specialized for any actual data management system (DMS), ranging from simple file managers (COBOL or RPG) to sophisticated DB managers such as relational systems and OO DBMS. These principles are organized as a DBRE methodology, but

2. There are two kinds of *Data Management Systems* (DMS), namely file management systems (FMS) and database management systems (DBMS).

we have tried to motivate and to justify them, in such a way that each reader can select and adapt those which are relevant to the problems (s)he has to cope with. This methodology can also be perceived as a *reference model* against which the proposals available in the literature and in vendor CASE tools and methodologies can be evaluated and compared.

The presentation is organized as follows. In Chapter 2, *Data schema specification*, we define a generic data structure description model in which conceptual, logical and physical structures can be specified accurately and reasoned on, among others through the concept of schema transformation. Chapter 3, *Database development revisited*, analyzes the database design process, emphasizing empirical practices and evaluating how the semantics defined in the conceptual schema is translated (or discarded) into the operational code. Chapter 4, *A general database reverse engineering methodology*, describes the processes of the proposed DBRE methodology. Two of them are the *Data structure extraction* (Chapter 5) and the *Data structure conceptualization* (Chapter 6) processes. The first one is dedicated to recovering the logical, or DMS-dependent, schema of the database while the second one addresses the problem of finding the conceptual structures underlying the logical schema. This generic methodology is specialized for some popular database managers in Chapter 7, (*DMS-oriented methodologies*), while CASE support is discussed in Chapter 8 (*CASE technology for DBRE*). A case study is developed in Chapter 9, *A case study: Database Migration*. Chapter 10, *Bibliographic notes and conclusion*, is devoted to the state of the art and to a open conclusion.

The reader is expected to have some basic knowledge in database management and design. References [Connolly 2002], [Elmasri 2000] and [Date 1995**???**] are suggested in data management, while [Batini 1992], [Teorey 1994**???**], [Elmasri 2000] and [Blaha 1998] are recommended in database design.

Chapter 2

Data schema specification

Abstract

Database reverse engineering mainly deals with schema extraction, analysis and transformation. In the same way as for any other database engineering process, it must rely on a rich set of models. These models must be able to describe data structures at different levels of abstraction, ranging from physical to conceptual, and according to various modeling paradigms. In addition, statically describing data structures is insufficient. We must be able to describe how a schema has evolved into another one. For instance, a physical schema leads to a logical schema, which in turn is translated into a conceptual schema. These transitions, which form the basis of DBRE, can be explained in a systematic way through the concept of schema transformation. In this section, we describe a data structure model with which schemas at different abstraction levels and according to the most common paradigms can be described precisely. Then, we present schema transformation operators that can explain inter-schema transitions.

2.1 A wide-spectrum specification model

In database development methodologies, the complexity is broken down by considering three abstraction levels. The engineering requirements are distributed among these levels, ranging from correctness to efficiency. At the conceptual level, the designer produces a technology-independent specification of the information, expressed as a *conceptual schema*, relying on an ad hoc formalism, called a conceptual model. At the logical level, the information is expressed in a model for which a technology exists. For instance, the required information is organized into a relational or object-oriented logical schema. Since reverse engineering is concerned with legacy systems, we will also consider CODASYL DBTG, IMS, TOTAL/IMAGE, COBOL, BASIC, RPG or ADABAS *logical schemas*. While a logical schema is based on a family of DMS models, a *physical schema* is dedicated to a specific DMS. In addition to the logical constructs, it includes technical specifications that govern data storage, access mechanisms, concurrency protocols or recovery parameters. We will talk about network logical schemas, but about, say, IDMS physical schemas.

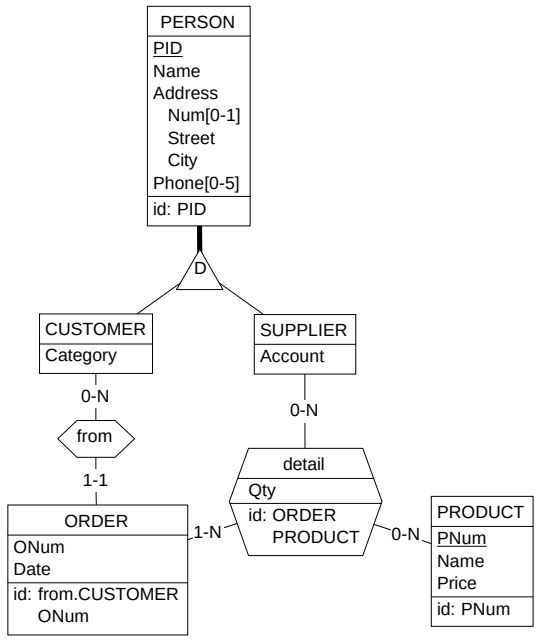
Due to the scope of this presentation, we cannot adopt specific formalisms for each of the abstraction levels. Instead, we will base the discussion on generic models that can easily be specialized into specific models. For instance, this model hierarchy can be translated into OMT/relational/ORACLE 8, into ERA/CODASYL/IDMS or into ORM/OO/O2 design methodologies. These models are derived from a unique model the formal basis of which has been developed in [Hainaut 1989].

Conceptual specifications

A conceptual schema mainly specifies entity types (or object classes), relationship types and attributes. Entity types can be organized into ISA hierarchies (organizing supertypes and subtypes), with total and disjoint properties. Attributes can be atomic or compound. They are characterized with a cardinality constraint [i-j] stating how many values can be associated with a parent instance (default is [1-1]). A relationship type has 2 or more roles. Each role also has a cardinality constraint [i-j] that states in how many relationships an entity will appear with this role. Entity types and relationship types can have identifiers, made up of attributes and/or remote roles. The source value set of an attribute can be a basic domain (e.g., numeric, boolean, character, time), a user-defined domain (e.g., VAT_number, Address, Pers_ID, URL) or an object class (in some OO models). Some of these constructs are illustrated in Figure 2-1.

In some situations, we will need more complex constraints, such as *existence constraints*, which state that a set B of components (attributes or remote roles) can be constrained by *co-existence* (either all elements of B must have a value or none), *exclusive* (at most one element of B can have a value) and *at-least-one* (at least one element of B must have a value) relations.

The exact terms used to denote these constructs will vary according to the specific model chosen. For instance, entity types will be called object classes in OMT or NOLOT in NIAM-like formalisms.



This schema includes entity types (or object classes) *PERSON*, *CUSTOMER*, *SUPPLIER*, *ORDER* and *PRODUCT*. *PERSON* has two disjoint subtypes, *CUSTOMER* and *SUPPLIER*. Relationship type *from* is binary while *detail* is ternary. Each *ORDER* entity appears in exactly 1 *from* relationship (cardinality [1-1]), and in at least one *detail* relationships (cardinality [1-N]).

Entity types and relationship types can be given attributes. For entity type *PERSON*, attribute *Name* is atomic, single-valued and mandatory. *Address* is a compound attribute. Its component *Num* is atomic, single-valued and optional (cardinality [0-1]). *Phone* is multivalued and optional: there are from 0 to 5 values per entity.

{*PID*} is the identifier of *PERSON*. The identifier of *ORDER* is made of external entity type *from.CUSTOMER* and of local attribute *ONum*.

There cannot exist more than one *detail* relationship with the same *ORDER* and *PRODUCT* entites.

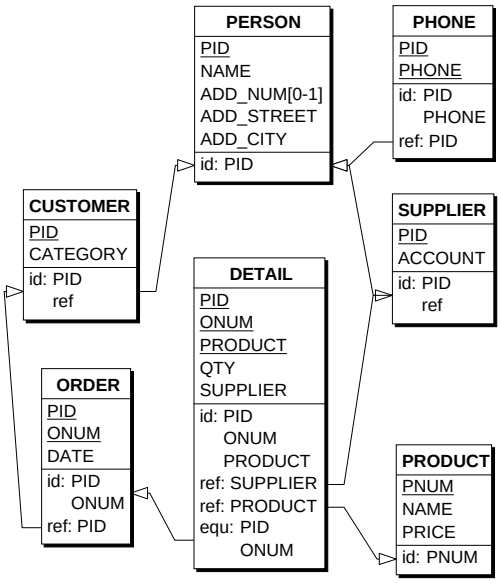
Figure 2-1: Conceptual structures.

Logical specifications

A logical schema comprises data structure definitions according to one of the commonly used families of models: relational model, network model (CODASYL DBTG), hierarchical model (IMS), shallow model (TOTAL, IMAGE), inverted file model (DATACOM/DB), standard file model (COBOL, C, RPG, BASIC) or object-oriented model. For instance, the schema of Figure 2-2 can be considered as a possible relational translation the conceptual schema of Figure 2-1. Similarly, one could have designed a network, hierarchical or OO equivalent schema. Since we want to discuss reverse engineering problems independently of the DMS model, we will use general terms such as *record type*, *inter-record relationship type* and *field*. For a specific model, these terms will translate into the specific terminology of this model. For instance, record types will be called *table* in relational schemas, *segment types* in hierarchical schemas, and *data sets* in shallow schemas. An *inter-record relationship* will be read *set type* in the network model, *access path* in the shallow model and *parent-child relationship* in the hierarchical model.

New constructs appear at this level, such as special kinds of multivalued attributes (bag, list,

or array), foreign keys and redundancy constraints. The equality constraint is a special kind of foreign key such that there exists an inverse inclusion constraint; see {*PID, ONUM*} of *DETAIL* in Figure 2-2.



This schema defines seven logical record types. *PERSON* has mandatory logical fields (*PID*, *NAME*, *ADD_STREET* and *ADD_CITY*) and one optional (nullable) field, *ADD_NUM*. Its primary identifier is {*PID*}. Field {*PID*} of *ORDER* is a foreign key to *CUSTOMER* (targeting its primary id). The group {*PID, ONUM*} of *DETAIL* is a multi-component foreign key. In addition, there is an inclusion constraint from {*PID, ONUM*} of *ORDER* to {*PID, ONUM*} of *DETAIL*. Combining these two constraints translates into an *equality* constraint (symbol **equ**). {*PID*} of *CUSTOMER* is both a primary id and a foreign key to *PERSON*.

Figure 2-2: Abstract logical structures. This relational schema is an approximate translation of the schema of Figure 2-1. The term record type must read *table*, field must read *column* and primary id must read *primary key*.

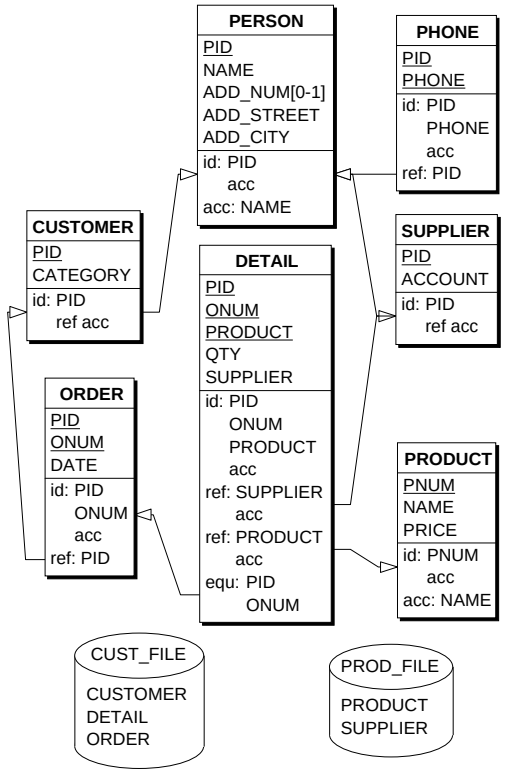
Physical specifications

Finally, physical specifications can be specified through a physical schema. Due to the large variety of DMS-dependent features, it is not easy to propose a general model of technical constructs. As far as reverse engineering is concerned, we will consider three essential concepts that may bring structural or semantic hints:

- *record collection* or *storage*, which is an abstraction of file, data set, tablespace, dbspace and any record repository in which data is permanently stored.
- *access key*, which represents any path providing a fast and selective access to records that satisfy a definite criterion; indexes, indexed set (DBTG), access path, hash files, inverted files, indexed sequential organizations all are concrete instances of the concept of access key;
- *cluster*, which is a physical grouping of records according to some logical relation in order to get fast access to these related records.

The first two constructs have been given a graphical representation (Figure 2-3). In database

design and development, other physical constructs can be of interest, such as page size, extent size, file size, buffer size, various fill factors, index technology, physical device and site assignment, etc. They will be ignored in this presentation.



This schema is made up of five physical record types and two record collections.
Collections *PROD_FILE* will store records of types *CUSTOMER*, *ORDER* and *DETAIL*.
Primary id {*PID*} of *CUSTOMER* is supported by an access key, denoted by symbol **acc**(ess key). The same holds for the other identifiers of the schema. An access key is also associated with some foreign keys (those which are not a prefix of another access key) and another one is defined on plain field {*NAME*}.
Other technical details are not shown in this graphical representation, such as record clusters, physical field sizes and coding schemes, page lengths, buffer management, index technologies and parameters, etc.

Figure 2-3: Abstract physical structures. This schema derives from the logical schema of Figure 2-2.

About schema modeling in DBRE

Since database reverse engineering starts from physical specifications and is to yield logical and conceptual schemas, it will naturally be based on the same models hierarchy as database development. An outstanding characteristic of database reverse engineering is that, most of the time, an in-progress schema includes physical, logical and conceptual constructs. Indeed, DBRE basically is a trial and error, exploratory and incremental process[1]. Therefore, while some parts of the schema may be, hopefully, conceptualized, other parts can be unprocessed

1. According to the standard taxonomy of process models, this activity can best be described as a *spiral* process.

so far, that is, only their physical constructs are elicited. This means that a data schema being reverse engineered must be described in a model that can **simultaneously** express physical, logical and conceptual constructs.

2.2 Schema transformation

It can be shown that almost all database engineering processes can be modeled as data structure transformations. Indeed, the production of a schema can be considered as the derivation of this schema from a (possibly empty) source schema through a chain of elementary operations called *schema transformations*. Adding a relationship type, deleting an identifier, translating names or replacing an attribute with an equivalent entity type, all are examples of basic operators through which one can carry out such engineering processes as building a conceptual schema [Batini 1992; Batini 1993], schema normalization [Rauh 1995], DBMS schema translation [Hainaut 1993b; Rosenthal 1988; Rosenthal 1994], schema integration [Batini 1992], schema equivalence [D'Atri 1984; Jajodia 1983; Kobayashi 1986; Lien 1982], data conversion [Navathe 1980], schema optimization [Hainaut 1993b; Halpin 1995] and others [Blaha 1996; De Troyer 1993; Fahrner 1995]. As will be shown later on, they can be used to reverse engineer physical data structure as well [Bolois 1994; Casanova 1984; Hainaut 1993b; Hainaut 1993a; Hainaut 1995].

Definition

A (schema) transformation is most generally considered as an operator by which a source data structure C is replaced with a target structure C'. Though a general discussion of the concept of schema transformation would include techniques through which new specifications are inserted (semantics-augmenting) into the schema or through which existing specifications are removed from the schema (semantics-reducing), we will mainly concentrate on techniques that preserve the specifications (semantics-preserving).

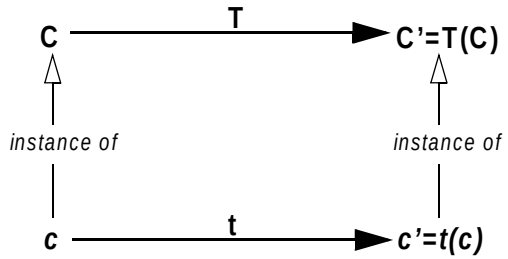


Figure 2-4: The two mappings of schema transformation $\Sigma \equiv <\mathbf{T},\mathbf{t}>$.

A transformation Σ can be completely defined by a pair of mappings <**T**,**t**> where **T** is called

the structural mapping and **t** the instance mapping. **T** explains how to replace C with C', while **t** states how instances of C must be replaced with instances of C' (Figure 2-4).

Another equivalent way to describe mapping **T** consists of a pair of predicates <**P**,**Q**>, where **P** is the weakest precondition C must satisfy for **T** being applicable, and **Q** is the strongest postcondition specifying the properties of C'. So, we can also write $\Sigma \equiv$ <**P,Q,t**>.

Semantics-preserving transformations

Each transformation $\Sigma_1 \equiv$ <$\mathbf{T}_1$,$\mathbf{t}_1$> can be given an inverse transformation $\Sigma_2 \equiv$ <$\mathbf{T}_2$,$\mathbf{t}_2$>, denoted Σ^{-1} as usual, such that, for any structure C,

$$\mathbf{P}_1(C) \Rightarrow C = \mathbf{T}_2(\mathbf{T}_1(C))$$

Σ_1 is said to be a **reversible transformation** if the following property holds, for any construct C and any instance c of C,

$$\mathbf{P}_1(C) \Rightarrow (C = \mathbf{T}_2(\mathbf{T}_1(C))) \wedge (c = \mathbf{t}_2(\mathbf{t}_1(c)))$$

So far, Σ_2 being the inverse of Σ_1 does not imply that Σ_1 is the inverse of Σ_2. Moreover, Σ_2 is not necessarily reversible. These properties can be guaranteed only for a special variety of transformations, called symmetrically reversible.

Σ_1 is said to be a **symmetrically reversible transformation**, or more simply **semantics-preserving**, if it is reversible and if its inverse is reversible too. Or, more formally, if both following properties hold, for any construct C and any instance c of C,

$$\mathbf{P}_1(C) \Rightarrow (C = \mathbf{T}_2(\mathbf{T}_1(C))) \wedge (c = \mathbf{t}_2(\mathbf{t}_1(c)))$$
$$\mathbf{P}_2(C) \Rightarrow (C = \mathbf{T}_1(\mathbf{T}_2(C))) \wedge (c = \mathbf{t}_1(\mathbf{t}_2(c)))$$

In this case, $\mathbf{P}_2 = \mathbf{Q}_1$ and $\mathbf{Q}_2 = \mathbf{P}_1$. A pair of symmetrically reversible transformations is completely defined by the 4-uple <$\mathbf{P}_1$,$\mathbf{Q}_1$,$\mathbf{t}_1$,$\mathbf{t}_2$>. Except when explicitly stated otherwise, all the transformations we will use in this presentation are semantics-preserving. In addition, we will consider the structural part of the transformations only.

Some popular semantics-preserving transformations

We propose in Figure 2-5 and Figure 2-6 two sets of the most commonly used transformational operators. The first one is sufficient to carry out the transformation of most conceptual schemas into relational logical schemas. The second comprises additional techniques particularly suited to derive optimized schemas. Experience suggests that a collection of about thirty of such techniques can cope with most database engineering processes, at all abstraction levels and according to all current modeling paradigms[2].

2. Provided they are based on the concept of record, entity or object.

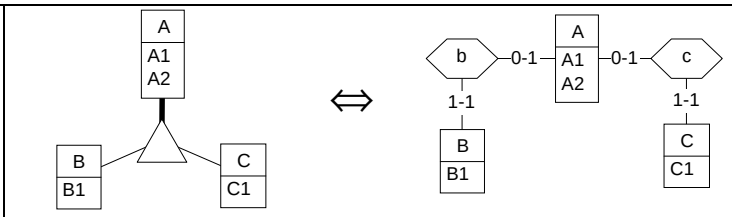
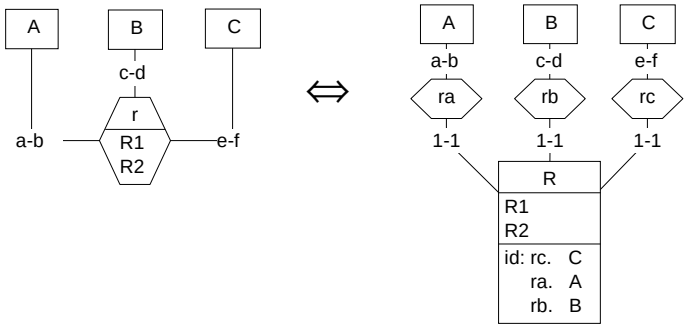
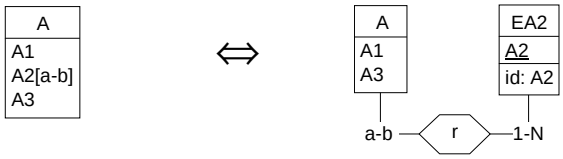
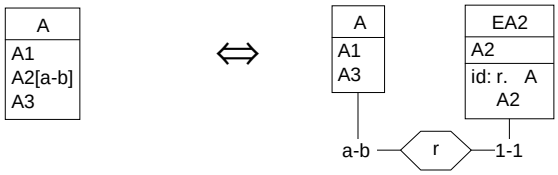
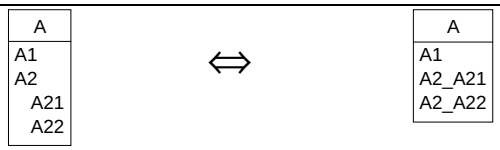
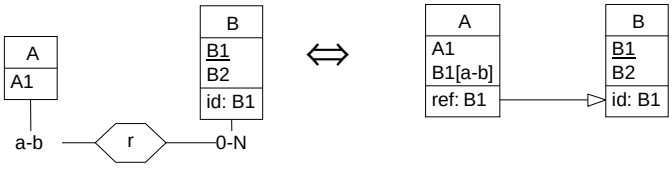
ISA-RT: Materializing an ISA relation into relationship type. *Inverse*: **RT-ISA**	
RT-ET: Transforming a rel-type into an entity type. *Inverse*: **ET-RT**.	
Att-ET/val: Transforming an attribute into an entity type (value representation). *Inverse*: **ET-Att**.	
Att-ET/inst: Transforming an attribute into an entity type (instance representation). *Inverse*: **ET-Att**.	
Disagg: Disaggregating a compound attribute. *Inverse*: **Aggreg**	
RT-FK: Transforming a binary relationship type into a foreign key. *Inverse*: **FK-RT**.	

Figure 2-5: Six major generic transformations with their inverse. Cardinalities a, b, c and d must be replaced with actual values.

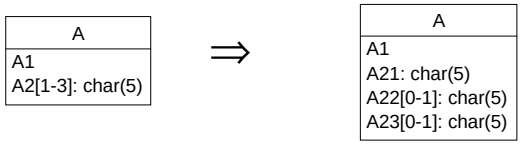
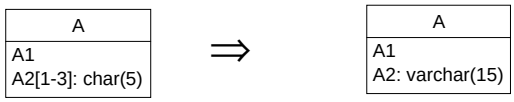
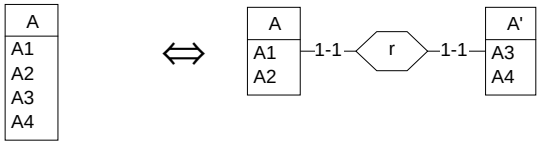
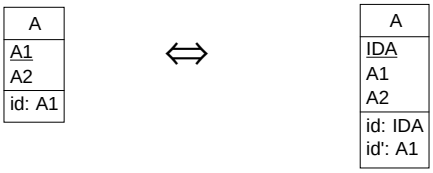
MultAtt-Serial: Replacing a multivalued attribute with a series of single-valued attributes that represents its instances. *Inverse*: **Serial-MultAtt**	
MultAtt-Single: Replacing a multivalued attribute with a single-valued attribute that represents the concatenation of its instances. *Inverse*: **Single-MultAtt**.	
Split: Transforming a binary relationship type into a foreign key. *Inverse*: **Merge**.	
AddTechID: A semantics-less attribute is added and made the primary ID of the ET. *Inverse*: **RemTechID**.	

Figure 2-6: Four additional generic transformations with their inverse. Though not indispensable in theory, they can prove useful, among others in optimization reasonings. As will be discussed later on, this simple form of MultAtt-Serial and MultAtt-Single are not semantics-preserving since right-side schemas include special forms of array.

Being functions, transformations can be composed in order to form more powerful operators. Complex transformation combinations can be built through **transformation plans**, which are high level semi-procedural scripts that describe how to apply a set of transformations in order to fulfil a particular task or to meet a goal. Figure 3-3 is a nice illustration of a simple transformation plan that explains how to transform a conceptual schema into relational structures. It is important to note that a transformation plan can be considered as a strategy for a higher level transformation Σ to be performed on a whole schema. The <P,Q> part of this transformation explains on whet kind of schemas Σ can be applied (P) and what will the resulting schema look like. In other words, in the script of Figure 3-3, P defines the Entity-relationship model while Q describes the relational model[3]. This analysis leads to an important conclusion for the following: *all engineering processes*, be they related to forward or reverse engineering, *can be considered as schema transformations*.

3. This observation is the basis of several components of the DB-MAIN CASE tool (Chapter 8) such the Schema Analysis assistant, the history manager and the method engine.

A practical application of semantics-preserving schema transformations

The four schemas of Figure 2-7 illustrate the step by step transformation of a source schema (*schema 1*) into an equivalent relational schema (*schema 4*) in which a partitioned ISA hierarchy has been reduced by upward inheritance. Since semantics-preserving transformations only have been used, both schema are semantically equivalent.

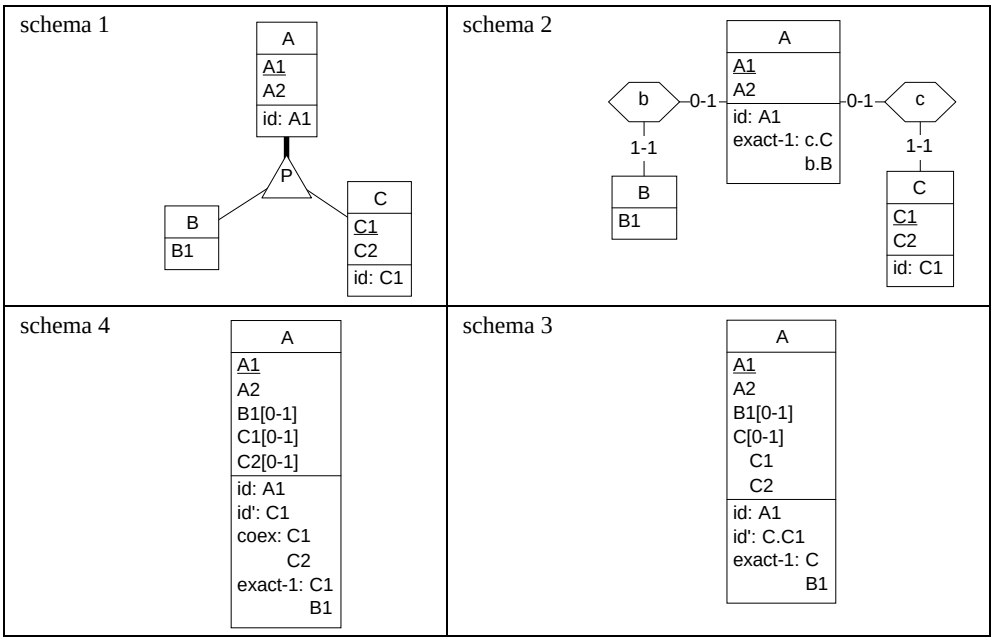


Figure 2-7: A complex schema transformation. First, the ISA relation is transformed through **ISA-RT**. Then, subtypes are transformed into attributes (**ET-Att**), which in turn are disaggregated (**Disagg**).

Chapter 3

Database development revisited

Each operational database D is the end product of a design process that starts with users requirement collection and that ends with code generation. What happens in between depends on the level of formality of the methodology followed by the developers and, *in fine*, on the maturity of the software engineering practices in the concerned development department. Whatever the way in which D was actually produced, we can imagine that it could have been obtained through a chain of standard design processes that translates in a systematic and reliable way the users requirements into this database as it appears currently. In the course of this chain, five important sets of documents would have been elaborated, namely the conceptual schema of D, its logical schema, its physical schema, the users views and the code that implements the data structures defined in the physical schema and in the users views. Of course, we all know that D was probably built by directly and incrementally encoding the DDL code according to the intuition and the expertise of the developer, and that these four outstanding documents never existed but in our mind. Nevertheless, trying to imagine what could have happened in this hypothetical design process can be a fruitful exercise since it will allow us to identify how and why the legacy database D has got to be what it currently is.

In this section, we will reexamine the database development process in order to identify the design decisions that shape legacy and current systems, and to understand how the intended semantics, *aka* users requirements, are implemented (or left unimplemented) in these systems and in their environments.

3.1 Database design framework

Most textbooks on database design of the eighties and early nineties propose a five step approach that is sketched in Figure 3-1. Through the *Conceptual Analysis* phase, the users requirements are translated into a conceptual schema, which is the formal and abstract expression of users requirements. The *Logical Design* phase transforms these abstract speci-

fications into data structures (the logical schema) that comply with the data model of a family of DMS. For instance, the conceptual schema is translated into relational, OO or standard file data structures. Through the *Physical Design* phase, the logical schema is augmented with DMS-specific technical specifications that make it implementable into a specific DMS and that gives it acceptable performance. From the logical schema, users views are derived that meet the requirements of classes of users (*View Design*). Finally, the physical schema and the users views are coded into the DDL of the DMS (*Coding*)

Three of these processes are worth being examined a bit further (Figure 3-2).

- *Conceptual Analysis* includes, among others, two major sub-processes, namely *Basic Analysis*, through which informal or semi-formal information sources are analyzed and their semantic contents are translated into conceptual structures, and (*Conceptual*) *Normalization*, through which these structures are given such additional qualities as readability, normality, minimality, extensibility, compliance with representation standards, etc. (see [Batini 1992] for instance).
- *Logical Design* may include, besides the translation phase, reshaping of the schema in order to make it meet technical or organizational requirements such as minimum response time, minimum disk space or low maintenance cost. We have called this process *Optimization*, since performance improvement is the most common objective. Some methodologies distinguish pre-translation optimization (sometimes called *Conceptual optimization*), which occurs before model translation, and in which the target DMS is ignored (see [Batini 1992] for instance) and post-translation optimization (*Logical Optimization*), which uses DMS-dependent reasonings and techniques.
- *Coding* can be more complex than generally presented in the literature and than carried out by CASE tools. Indeed, any DMS can only cope with a limited range of structures and integrity constraints for which its DDL provides an explicit syntax. For instance, a SQL-2 DBMS knows about machine value domains, unique keys, foreign keys and mandatory columns only. If such constructs appear in a physical schema, they can be explicitly declared in the SQL-2 script. On the contrary, all the other constraints must be either ignored or expressed in any other way (at best through *check* predicates or *triggers*, but more frequently through procedural sections scattered throughout the application programs). So we must distinguish two kinds of operational code: $\mathbf{code_{ddl}}$, which comprises the explicit declaration of constructs, expressed in the DDL of the DMS and $\mathbf{code_{ext}}$ that expresses, in more or less readable and identifiable form, all the other constructs.

From now on, we will ignore the *Conceptual Analysis* phase, which has no impact on reverse engineering, except for the *Normalization* process, which will also be useful as a finishing touch. The other two processes will be discussed in more detail in the following sections.

Though database design has been one of the major theoretical and applied research domains in software engineering in the last two decades, and though it can be considered as a fairly well mastered problem, we are faced here with files and database that have not been built according to well structured methodologies. Tackling the reverse engineering of a database needs a deep understanding of the forward process, i.e., database design, not only according

to standard and well formalized methodologies, but above all when no such rigorous methods, or on the contrary more sophisticated ones, have been followed. In other words, we intend to grasp the mental rules that govern the intuitive and empirical behavior of practitioners. The approach should not be normative, as in forward engineering, where practitioners are told how to work, but rather descriptive, since we have to find out how they have worked.

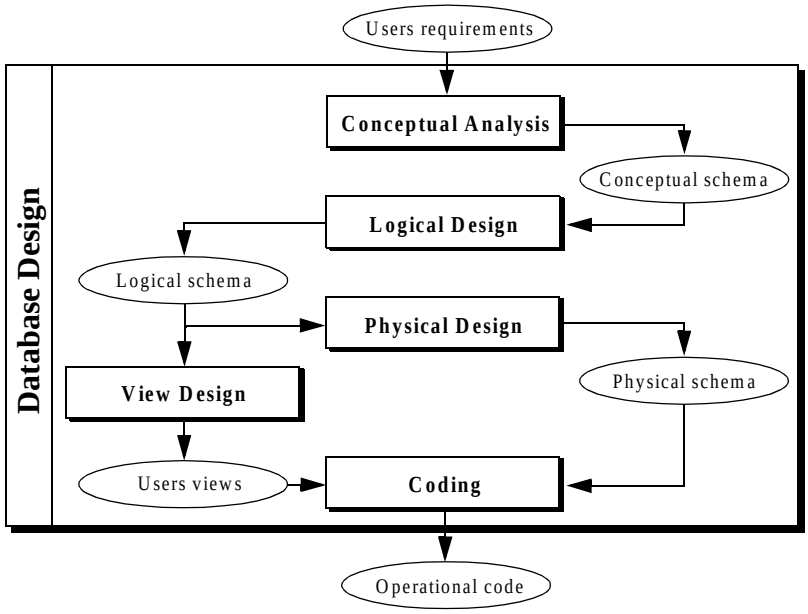


Figure 3-1: Main products and processes of standard database design methodologies.

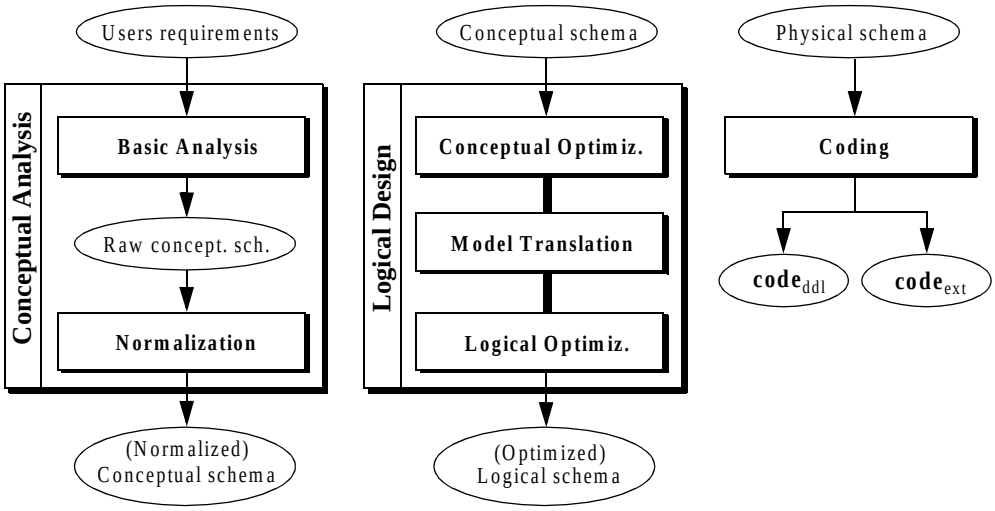


Figure 3-2: Common strategies for Conceptual Analysis, Logical Design and Coding.

1. for each supertype **E**, do:
   apply **Mat-ISA** to **E**;
2. for each non-functional relationship type **R**, do:
   apply **RT-ET** to **R**;
3. while compound or multivalued attributes still exist, do:
   for each top-level attribute ***A*** that is both single-valued and compound, do:
   apply **Disaggr** to ***A***;
   for each top-level attribute ***A*** that is multivalued, do:
   apply **Att-ET/inst** to ***A***;
4. until no relationship types can be transformed, repeat:
   for each relationship type **R**, do:
   if **R** meets the precondition of **RT-FK**, apply **RT-FK** to **R**;
5. until no relationship types remain, repeat:
   for each relationship type **R**, do:
   apply **AddTechID** to the entity type of **R** where an identifier is missing;
   do 4;

Figure 3-3: Sample transformation plan for translating a conceptual schema into a relational logical schema. This plan is a suggested strategy for the *Model Translation* process (Figure 3-2)

3.2 Transformational description of the *Logical design* process

Most of the processes that make database design methodologies can be described as schema transformations, that can in turn be refined, directly or indirectly, into lower-level processes or operators that are primitive schema transformations as defined in Section 2.2. Such is the case for the logical design phase, which ideally is a pure semantic-preserving transformation of the conceptual schema into a DMS-compliant optimized logical schema. As an illustration, the translation procedure of Figure 3-3, based on the semantics-preserving transformations of Figure 2-5, can be a simple strategy for the *Model Translation* process specialized for standard relational structures. We will call this sort of procedures a *transformation plan*. It can be seen as the strategy implementing a higher-lever transformational process, here *Relational Logical design*. Transformation plans can be developed for every DMS. However, it must be taken into account that such a strategy is ideal, in that it does not address any criteria other than RDBMS compliance and semantics preservation. Empirical strategies also depend on criteria such as performance (see below), and generally are more complex. Nevertheless, each of them can be defined in terms of a limited set of primitive schema transformations.

Modeling the major database design processes as symmetrically reversible transformations is important because it naturally lead to the transformational modeling of the inverse process, i.e. database reverse engineering. Indeed, reversing the order of the operations and replacing each of them with its inverse can be a good starting point for a reverse engineering strategy. Though things will prove a bit more complex than considered so far, this framework will be adopted in the following.

3.3 Optimization techniques

Very few legacy systems have been developed without optimization. Identifying the optimization structures that have been introduced is important in order to discard them when trying to recover the semantics of the data structures. In particular, knowing what were the technical constraints, the main optimization criteria, the development culture and the programmer's skills can prove essential to interpret correctly large and complex schemas.

Several optimization criteria can be considered, either independently or simultaneously: secondary memory space, main memory space, response time, update time, transaction throughput, exploitation time, portability, maintenance cost, evolution cost are some examples. Few studies have proposed a comprehensive and generalized treatment of the problem, so that in most cases, addressing this issue will be based on the developer's expertise and on some vague recommendations provided by the technical manuals of the DBMS. We will briefly discuss four classes of optimization techniques.

Discarding constructs

Intentionally ignoring a costly construct can be a good way to decrease the overall cost of the system. Such a practice can lead to leaving aside identifiers and foreign keys. Unrealistic constraints defined at the conceptual level (such as *the average amount of expenses per department cannot increase of more than 5% per month*) will generally be dropped at the logical level. Since there will be no formal trace of these constructs, they could be recovered through data analysis techniques or through examination of the system environment only.

Structural redundancy

Structural redundancy techniques consist in adding new constructs in a schema such that their instances can be computed from instances of other constructs. Attribute duplication, relationship type composition and aggregated values (such as *count*, *sum*, *average*) representation are some examples of common optimization structural redundancies. These transformations are (trivially) symmetrically reversible since they merely add derivable constructs without modifying the source constructs. The reverse transformation consists in removing the redundant constructs. The main problem is to detect the redundancy constraint that states the equivalence of construct.

Normalization redundancy

The most common unnormalization technique are based on merging two entity types linked by a one-to-many relationship type into a single entity type. This technique allows obtaining the information of both entity types in one logical access, thereby decreasing the access time. On the other hand, it induces redundancies. An unnormalized structure is detected in entity type B by the fact that the determinant of a dependency F is not an identifier of B.

Restructuration

Restructuration consists in replacing a construct with other constructs in such a way that the resulting schema yield better performance. These techniques introduce no redundancy.

Four popular techniques are vertical partitioning and horizontal partitioning of entity types as well as their inverse.

- Horizontal partitioning consists in replacing entity type A with entity types A1 and A2 of identical structure in such a way that the population of A is partitioned into those of A1 and A2. This technique yield smaller data sets, smaller indexes and allows for better space allocation and management (e.g., backup and distribution). Horizontal partitioning can be applied to relationship types as well. For instance the population of many-to-many relationship type R can be split into one-to-many relationship type R1 that collects selected instances, while many-to-many relationship type R2 collects the others. The implementation of R1 (as a foreign key) can be more efficient than that of R2 (through a relationship table).

- Its inverse, i.e. horizontal merging, decreases the number of entity types and makes physical clustering easier.
- Vertical partitioning of entity type A partitions its attribute/role components into two (or more) entity types A1 and A2, linked by a one-to-one relationship type (transformation **Split**). This partitioning is driven by processing considerations: components that are used simultaneously are grouped into a specific entity type. This decreases the physical length of A entities, and improves access time.
- Vertical merging is the inverse technique (transformation **Merge**). It consists in merging two entity types that are linked by a one-to-one relationship type, or equivalently by an identifying foreign key, in order to avoid double access to get related entities.

Of course, there are many other techniques that can yield improvement according to some performance criteria. We mention some of them.

A frequent restructuration consists in replacing some one-to-many relationship types by foreign keys (**RT-FK**), even in DBMS supporting such relationship types (IMS, CODASYL). The argument can be to avoid complex structures (e.g., logical relationships in IMS databases) or to physically split the database into independent fragments that are easier to distribute, to recover or to backup (e.g., in CODASYL databases).

Implementing a large size attribute as a reference entity type (**Att-ET**) can reduce disk space when coupled with the **AddTechID** transformation.

Implementing a dependent entity type as a multivalued, compound attribute integrated into the parent entity type (**ET-Att**) can improve access time (Figure 6-3, reverse).

We will mention a last restructuration technique, namely **AddTechID**, through which a long and/or multi-component primary identifier is replaced with a short, meaningless, attribute. This will reduce the size of the primary index, thus improving the access time, but this will also reduce the complexity and the size of the foreign keys that include copies of this identifier. The former primary identifier gets the secondary status, in such a way that no semantics is lost.

3.4 Expressing and coding non-DMS constructs

Since these specifications are ignored by the DMS, their expression, called $\mathbf{code_{ext}}$ in Figure 3-2, is up to the developer/programmer, and therefore exhibits a very large variety of coding schemes, even when one developer only is involved. However, due to the *forensic nature* of reverse engineering activities, it is not completely useless to recall some of the most popular ways to express non-DMS constructs. We will distinguish two categories of translation techniques, namely **internal** and **external** (*vis à vis* the DMS). To make things more concrete, we consider that the non-DMS construct is integrity constraint **IC**.

Internal techniques

They use special programming facilities of the DMS to encode either IC itself, or the procedural fragments that ensure the satisfaction of IC. We will mention four of them, that specifically appear in RDBMS and in some OO-DBMS. It is fairly easy to locate this type of code and to identify the construct it implements.

- **Predicates**. Constraint IC can be expressed as a table predicate (`check`) or as a schema predicate (`assertion`). For each event that modifies the contents of the table(s) mentioned in the predicate, the DMS will ensure that the update will not violate the predicate, otherwise the operation is cancelled.
- **Views with check option**. If a `view` is defined on a single table and if it includes a `where` part that encodes IC, this view yields the consistent rows only, i.e., those which satisfy IC. The additional clause `with check option` instructs the DMS to filter the insert and update operations as well, in such a way that only consistent rows can appear in the base table.
- **Triggers**. This *event-condition-action* mechanism allows the developer to define the behavior of the DMS when it is asked to insert, delete or update rows in a specific table. The triggering operation can be aborted or compensated in such a way that the final state of the database will always be consistent according to IC.
- **Stored procedures**. These centralized procedures are associated with the database and can be invoked from any process: user interface, application programs, triggers, other stored procedures. They can be developed as the unique API through which the programmer can update the database (no direct privileges are granted for `insert`, `delete` and `update` primitives). Each of these procedures will include all the code necessary to ensure that the integrity constraints, included IC, are satisfied.

External techniques

They resort to code fragments in the application programs to encode the management of IC. Detecting the constructs they implement can be very complex and tedious, and makes one of the most challenging problems of reverse engineering. Unfortunately, they are far more frequent than internal techniques. We will describe five of them.

- **Access module**. Each application program uses the services of an API that is in charge of managing the data, including data integrity control. Constraints such as IC are taken in charge through ad hoc code sections that ensure that no actions called for by the application program can corrupt the data. This API is developed as a general purpose library by the programmer. This technique is often used for masking standard file management idiosyncrasies, therefore improving the portability of programs. This clean architecture is very helpful when reverse engineer the data structures.
- **Code fragments scattered throughout the procedural code**. This approach encompasses several techniques. Their common point is that a specific code section is inserted before each data modification statement that may lead to the violation of IC. The main

problems are the high degree of duplication of this code and the unavoidable discrepancies among all these sections, particularly when the programs have been written by different persons at different periods. This practice, which unfortunately is the most widespread, makes reverse engineering, as well as maintenance, particularly complex.

- **User interface validation code fragments**. Validation routines that check that IC cannot be violated are associated with dialog objects in the user interface. This was the privileged way of managing data integrity in early RDBMS such as Oracle 5, through its dialog manager SQL-Forms. This technique was adequate for interactive transactions, but left data unprotected against direct DML calls from application programs. However, it gives reliable hints for reverse engineering.
- **Pre-validation programs**. Candidate data modifications are batched in a modification file. Then, at regular interval, they are checked against IC (and other constraints). The modifications that satisfy IC are introduced in the database. This primitive but efficient technique, as well as the next one, is very helpful for reverse engineering.
- **Post-validation programs**. All the modifications are carried out unvalidated. At definite time points, the data are checked, and those which violate IC are removed from the database.

3.5 Analysis of semantics preservation in empirical database design

The final, operational description of the database can be seen as the result of a chain of transformations that have *degraded* the origin conceptual schema. This degradation takes two forms. Firstly, some semantics originally expressed in the conceptual schema has been *discarded* at different levels of the whole process, thus yielding an incomplete implementation. Secondly, the final form naturally is *less readable* than its conceptual source and therefore will be more difficult to interpret.

The origin of schema degradation

Let us reexamine each process of database design as far as it introduces some degradation of the conceptual specifications.

- *Logical design.* This phase may introduce the two forms of degradation to a large extent.
  - *Conceptual and logical optimization.* The schema is restructured according to design requirements concerning access time, distribution, data volume, availability, etc. It is obscured due to non-semantic restructuration, such as structure splitting or merging, denormalization or structural redundancies. These processes can deeply change the shape of the logical schema, but, ideally, they should not change its information contents. However, we have seen that intentionally discarding constructs can be a way to lower the cost of the system.

- *Model translation*. The data structures are transformed in order to make them compliant with the model of the target DMS. Generally, this process deeply changes the appearance of the schema in such a way that the latter is still less readable. For instance, in standard files (resp. relational DBMS) many-to-many relationship types are transformed into record types (tables) while many-to-one relationship types are transformed into reference fields (foreign key). In a CODASYL schema, a secondary identifier is represented by an indexed singular set. In a TOTAL or IMAGE database, a one-to-many relationship type between two major entity types is translated into a detail record type. Frequently, names have to be converted due to the syntactical limitations of the DMS or of the host languages.
  On the other hand, due to the limited semantic expressiveness of older (and even current) DMS, this translation is seldom complete. It produces two subsets of specifications: the first one being strictly DMS-compliant while the second one includes all the specifications that cannot be taken in charge by the DMS. For instance, referential constraints cannot be processed by standard file managers. In principle, the union of these subsets completely translates the conceptual specifications.

- *Physical design*. Operates at the internal level, and does not modify the schema as it is seen by the programmer. However, physical constructs often are declared with the same DDL as logical constructs, so that the DDL code can be obstructed with physical declaration. This particularly true with older DBMS, such CODASYL and IMS, for which the borderline between logical and physical concepts was still unclear.
- *Coding*. Only the subset of the physical schema that is strictly DMS-compliant can be translated into the DDL of the DMS (**code$_{ddl}$**). The discarded specifications are either ignored, or translated into external languages, systems and procedures that push the coded constructs out of control of the DMS (**code$_{ext}$**).
  Another phenomenon must be pointed out, namely **structure hiding** (Figure 3-4). When translating a data structure into DMS-DDL, the developer may choose to hide some information, leaving to the host language the duty to recover it. A widespread example consists in declaring some subset of fields (or even all the fields) of a record type as a single, unstructured, field; recovering the hidden structure can be made by storing the unstructured field values into a host program variable that has been given the explicit structure. Finally, let's observe that the DMS-DDL schema is not always materialized. Many standard file managers, for instance, do not offer any central way to record the structure of files, e.g., in a data dictionary.

Intended record structure

```
01 CUSTOMER.
   02 C-KEY.
      03 ZIP-CODE pic X(8).
      03 SER-NUM  pic 9(6).
   02 NAME     pic X(15).
   02 ADDRESS  pic X(30).
   02 ACCOUNT  pic 9(12).
```

Coded record structure

```
01 CUSTOMER.
   02 C-KEY  pic X(14).
   02 filler pic X(57).
```

Figure 3-4: An example of *structure hiding*. The decomposition of both the key part and the data part is replaced with anonymous data structures in the actual coded structure. Though it can simplify data description and data usage, this frequent technique makes data structure considerably more complex to recover.

- *View design*. Since a view merely describes derived data from its logical description, it should not contribute to the problem of semantics preservation/degradation. However, as illustrated in Figure 3-5, a view can express constructs that have not been preserved in the operational code. Therefore, it will be an essential source of information when we try to recover constructs discarded from the operational code.

```
create table CUSTOMER(CID numeric(8),NAME char(24),ADDRESS char(80));
create view CUSTOMERS(CID, NAME, STREET, CITY) as
  select CID,NAME,substr(ADDRESS,7,32),substr(ADDRESS,39,24) from CUSTOMER;
```

Figure 3-5: A view can express richer data structures than the DDL code that implements the global physical schema.

Transformational analysis of empirical designs

Ignoring the conceptual phase, which is of no interest for DBRE, as well as users views derivation, database forward engineering can be modeled by (the structural part of) transformation **FE**:

code = **FE**(**CS**)

where **code** denotes the operational code and **CS** the conceptual schema.

Denoting the logical and physical schemas by **LS** and **PS** and the logical design, physical design and coding phases by **LD**, **PD** and **COD**, we can refine the previous expression as follows:

LS = **LD**(**CS**)
PS = **PD**(**LS**)
code = **COD**(**PS**)

We consider function σ(**D**) which gives the semantics[1] of document **D**. Ideally, we should have

$$\sigma(\mathbf{code}) = \sigma(\mathbf{PS}) = \sigma(\mathbf{LS}) = \sigma(\mathbf{CS}),$$

but we know that empirical processes do not meet this property. Let us call Δ the semantics of **CS** that disappeared in the process. We get:

$$\sigma(\mathbf{code'}) \cup \Delta = \sigma(\mathbf{CS})$$

where **code'** denotes the *actual* code.

As we learned in this section, Δ has been lost at several levels. This means that Δ should be partitioned into $\Delta = \Delta l \cup \Delta p \cup \Delta c$, where Δl denotes the conceptual specifications left aside during **LD**, Δp the semantics ignored in **PD** and Δc the semantics discarded in **COD**.

Let us call **LS'** the partial logical schema resulting from **LD** and **PS'** the actual result of **PD**. We have:

$$\sigma(\mathbf{LS'}) \cup \Delta l = \sigma(\mathbf{LD}(\mathbf{CS}))$$
$$\sigma(\mathbf{PS'}) \cup \Delta p = \sigma(\mathbf{PD}(\mathbf{LS'}))$$
$$\sigma(\mathbf{code'}) \cup \Delta c = \sigma(\mathbf{COD}(\mathbf{PS'}))$$

Finally, we can refine this analysis by considering that:

1. **code** can be split into its DDL part and its external part: $\mathbf{code'} = \mathbf{code_{ddl}'} \cup \mathbf{code_{ext}'}$
2. accordingly, process **COD** comprises two subprocesses[2]: $\mathbf{COD} \equiv \{\mathbf{COD_{ddl}}; \mathbf{COD_{ext}}\}$
3. process **LD** comprises three subprocesses, namely conceptual optimization, model translation and logical optimization; to simplify things, we will merge both optimization processes: $\mathbf{LD} \equiv \{\mathbf{OPT}; \mathbf{TRANS}\}$
4. the discarded specifications Δ have not disappeared, but rather are hidden somewhere in the operational system or in its environment. For instance, they can be discovered in the data themselves (through data analysis techniques) or found in the exploitation and users procedures (and can be elicited by interviewing users, exploitation engineers or developers). Let us call $\mathbf{E(\Delta)}$ the observable information sources, such as file contents, environment behavior and users knowledge, in which we can find traces of Δ.

Now we can complete the description of the semantics properties of forward processes. For

---

1. We can leave the concept of semantics undefined and base the discussion on its intuitive interpretation. Considering the goal of the conceptual schema, and adopting the conceptual formalism as a pure expression of all the semantics of the system and only it, we could write: $\sigma(CS) = CS$. However, due to the fact that several conceptual schemas can describe the same application domain (a fact sometimes called semantic relativism), we will distinguish CS as a document from its semantic contents $\sigma(CS)$.
   If needed, we can consider that the semantics of a document is a consistent set of logic assertions such that if $D1 \subseteq D2$, then $\sigma(D2) \Rightarrow \sigma(D1)$. To simplify the discussion, we will consider that the semantics is expressed in such a way that we can also write: $\sigma(D1) \subseteq \sigma(D2)$. In addition this formalism must be such that $\sigma(D1 \cup D2) = \sigma(D1) \cup \sigma(D2)$.
2. The notation $P \equiv \{P1;P2\}$ means that carrying out process P1 implies carrying out P1 and P2 in any order. In the same way, $P \equiv P2 \circ P1$ means that P consists in carrying out P1 first, then P2 on the result just obtained.

ideal approaches, we have:

$$\mathbf{code} = \mathbf{FE}(\mathbf{CS})$$
$$\mathbf{FE} \equiv \{\mathbf{COD_{ddl}}; \mathbf{COD_{ext}}\} \circ \mathbf{PD} \circ \{\mathbf{OPT}; \mathbf{TRANS}\}$$

or

$$\mathbf{LS} = \{\mathbf{OPT}; \mathbf{TRANS}\}(\mathbf{CS})$$
$$\mathbf{PS} = \mathbf{PD}(\mathbf{LS})$$
$$\mathbf{code_{ddl}} = \mathbf{COD_{ddl}}(\mathbf{PS})$$
$$\mathbf{code_{ext}} = \mathbf{COD_{ext}}(\mathbf{PS})$$
$$\mathbf{code} = \mathbf{code_{ddl}} \cup \mathbf{code_{ext}}$$

with $\sigma(\mathbf{code}) = \sigma(\mathbf{code_{ddl}}) \cup \sigma(\mathbf{code_{ext}}) = \sigma(\mathbf{PS}) = \sigma(\mathbf{LS}) = \sigma(\mathbf{CS})$,

For empirical approaches, we have:

$$\mathbf{code'} \cup \mathbf{E}(\Delta) = \mathbf{FE}(\mathbf{CS})$$

or

$$\mathbf{LS'} \cup \mathbf{E}(\Delta l) = \{\mathbf{OPT}; \mathbf{TRANS}\}(\mathbf{CS})$$
$$\mathbf{PS'} \cup \mathbf{E}(\Delta p) = \mathbf{PD}(\mathbf{LS'})$$
$$\mathbf{code_{ddl}'} \cup \mathbf{E}(\Delta c) = \mathbf{COD_{ddl}}(\mathbf{PS'})$$
$$\mathbf{code_{ext}'} = \mathbf{COD_{ext}}(\mathbf{PS'})$$
$$\mathbf{code'} = \mathbf{code_{ddl}'} \cup \mathbf{code_{ext}'}$$

with
$$\sigma(\mathbf{code'}) \cup \Delta = \sigma(\mathbf{code_{ddl}'}) \cup \sigma(\mathbf{code_{ext}'}) \cup \Delta c \cup \Delta p \cup \Delta l$$
$$= \sigma(\mathbf{PS'}) \cup \Delta p \cup \Delta l$$
$$= \sigma(\mathbf{LS'}) \cup \Delta l$$
$$= \sigma(\mathbf{CS})$$

So, we have redefined empirical database forward engineering as a reversible transformation made up of reversible sub-processes that losslessly transform their input specifications into equivalent output specifications.

Chapter 4

A general database reverse engineering methodology

Abstract

The problems that arise in database reverse engineering naturally fall in two categories that are addressed by the two major processes in DBRE, namely *data structure extraction* and *data structure conceptualization*. By *naturally*, we mean that these problems relate to the recovery of two different schemas (resp. logical and conceptual), and that they require quite different concepts, reasonings and tools. In addition, each of these processes grossly appears as the reverse of a standard database design process (resp. coding/physical design and logical design). The *Data structure extraction* process recovers the complete logical schema, including the explicit and implicit structures and properties. The *Data structure conceptualization* process recovers the conceptual schema underlying this logical schema.

In this section, we will first derive a transformational model of DBRE, then we will describe the contents and the objective of the proposed database reverse engineering methodology. The specific techniques and reasonings of each process will be developed in Chapters 5 and 6.

4.1 Deriving a transformational DBRE model

Since reverse engineering consists in recovering (among others) the conceptual schema from the operation code, it is reasonable to consider that this process is just the reverse of forward engineering. Though things will prove a bit more complex, this hypothesis is a good starting point.

$$\mathbf{CS} = \mathbf{FE}^{-1}(\mathbf{code})$$

We first observe that reversing a hierarchically decomposable process consists in inverting the order of the sub-processes, then replacing each sub-process with its inverse. For systems that have been developed according to an ideal approach, we have:

$$\mathbf{CS} = \mathbf{FE}^{-1}(\mathbf{code})$$
$$\mathbf{FE}^{-1} \equiv \{\mathbf{OPT}^{-1}; \mathbf{TRANS}^{-1}\} \circ \mathbf{PD}^{-1} \circ \{\mathbf{COD_{ddl}}^{-1}; \mathbf{COD_{ext}}^{-1}\}$$

or

$$\mathbf{PS_{ddl}} = \mathbf{COD_{ddl}}^{-1}(\mathbf{code_{ddl}})$$
$$\mathbf{PS_{ext}} = \mathbf{COD_{ext}}^{-1}(\mathbf{code_{ext}})$$
$$\mathbf{PS} = \mathbf{PS_{ddl}} \cup \mathbf{PS_{ext}}$$
$$\mathbf{LS} = \mathbf{PD}^{-1}(\mathbf{PS})$$
$$\mathbf{CS} = \{\mathbf{OPT}^{-1}; \mathbf{TRANS}^{-1}\}(\mathbf{LS})$$

Grossly speaking, the suggested DBRE approach comprises three steps:

1. recovering the physical schema **PS** from the operational code {$\mathbf{code_{ddl}}$, $\mathbf{code_{ext}}$}, i.e., *undoing* the Coding forward process; this consists in *uncoding* the DDL code, then *uncoding* the non-DDL code and finally merging them into **PS**; we will call this process *Extraction* of **PS**
2. recovering the logical schema **LS** from the physical schema **PS**, i.e., *undoing* the physical design forward process; this should be fairly simple since the forward process consists in adding technical constructs to the logical structures;
3. recovering the conceptual schema **CS** from the logical schema **LS**, i.e. removing and transforming the optimization constructs (what we will call *de-optimizing* **LS**) and interpreting the logical constructs in terms of their source conceptual structures (we will call this *unstranslation* of **LS**); the whole step will be called *Conceptualization* of **LS**.

Now, we consider applications that have been developed according to some empirical approach. The main difference is that a part of the semantics called (Δ) may lie outside the system (in **E(Δ)**), i.e., it has not be wired in the coded part of this system. This semantics can be found elsewhere, for instance in the environment of the application or in the data. As a consequence, we must base the reverse engineering process on incomplete code ($\mathbf{code_{ddl}}$**'** and $\mathbf{code_{ext}}$**'**) and on **E(Δ)**. The fact that the missing semantics has been lost at different levels is irrelevant. Therefore, we will consider that analyzing **E(Δ)** is a task of the extraction process. The overall process can still be refined according to the following remarks.

- We observe that *undoing* the physical process is a fairly trivial task that can be integrated as a sub-process of the *Extraction* step; it will be called *Schema Cleaning*.

- It will appear in the following that the extraction of **code$_{ddl}$'** is an easy task while the extraction of specifications from **code$_{ext}$'** and **E(Δ)** is much more complex; we suggest to call the analysis of **code$_{ddl}$'** *DDL code Extraction* while the analysis of {**code$_{ext}$'**; **E(Δ)**} will be called *Schema refinement*.
- The *Conceptualization* step works on a logical schema that may include awkward constructs, such as inconsistent names, that will first be processed through a sub-process called *Schema Preparation*.
- The *Conceptualization* step yields a conceptual schema that can include awkward constructs too. Retructuring this conceptual schema in order to make it look better according to corporate standards will be called *Conceptual Normalization*.

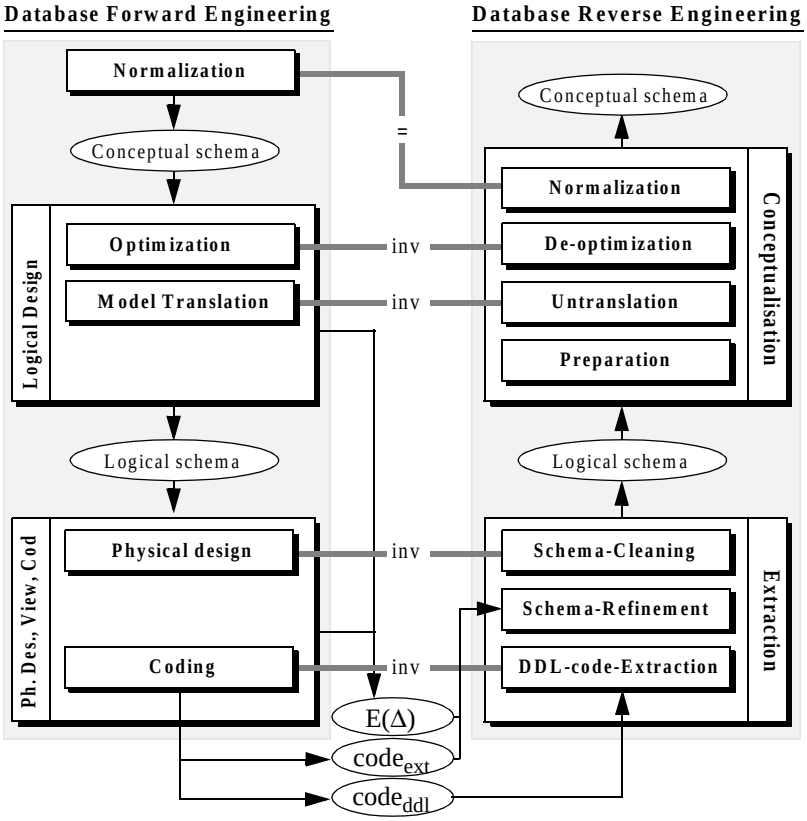


Figure 4-1: The main DBRE processes as the inverse of forward processes. Symbol **inv** on the forward/reverse correspondences means that each process is the inverse of the other, while symbol **=** indicates they are the same.

We can now propose a generic database reverse engineering framework that takes into ac-

count empirical designs.

CS = **RE**(**code'** ∪ **E(Δ)**)
RE ≡ **CONCEPTUALIZATION**o **EXTRACTION**
EXTRACTION ≡ **Schema-Cleaning**o **Schema-Refinement**o **DDL-code-Extraction**
CONCEPTUALIZATION ≡
  Normalizationo {**De-optimization**;**Untranslation**}o **Preparation**

Logical-schema = **EXTRACTION**(**code'** ∪ **E(Δ)**)

Conceptual-schema = **CONCEPTUALIZATION**(**Logical-schema**)

Explicit-physical-schema = **DDL-code-Extraction**($\mathbf{code_{ddl}}$')
Complete-physical-schema =
  Schema-Refinement(**Explicit-physical-schema**, $\mathbf{code_{ext}}$'∪**E(Δ)**)
Logical-schema = **Schema-Cleaning**(**Complete-physical-schema**)

Raw-conceptual-schema =
  {**De-optimization**; **Untranslation**}o **Preparation**(**Logical-schema**)
Conceptual-schema = **Normalization**(**Raw-conceptual-schema**)

This discussion has ignored the coded users views, which can prove an important information source. They will be introduced in the detailed description of the *Extraction* process.

The framework we have just built can be sketched graphically in order to show the links with the forward processes (Figure 4-1).

4.2 The major phases of the DBRE methodology

The general architecture of the reference DBRE methodology is outlined in Figure 4-2. It shows clearly the two main processes that will be described in Chapters 5 and 6.

Experience teaches that managing the whole project and identifying the relevant information sources is not an easy task. Therefore we add a preliminary step called ***Project Preparation***, which aims at identifying and evaluating the components to analyze, at evaluating the resources needed and at defining the operations planning. Its main sub-processes are (1) *System identification*, through which the files, programs, screens, reports, forms, data dictionaries, repositories, program sources and documentation are identified and evaluated, (2) *Architecture recovery*, that consists in drawing the main procedural and data components of the system and their relationships, (3) *Identification of the relevant components*: all the components that bring no information are discarded, (4) *Resource identification*, in terms of skill, work force, calendar, machine, tools and budget, and finally (5) *Operation planning*. We will not develop this point, for which the reader is suggested to consult references such as [Aiken 1996], which develops various aspects of DBRE project management.

The ***Data structure extraction*** process aims at rebuilding a complete logical schema in which all the explicit and implicit structures and properties are documented. The main source of

problems is the fact that many constructs and properties are implicit, i.e. they are not explicitly declared, but they are controlled and managed through, say, procedural sections of the programs. Recovering these structures uses DDL code analysis, to extract explicit data structures, and data structure elicitation techniques, that lead to the recovery of implicit constructs.

The ***Data structure conceptualization*** process tries to specify the semantic structures of this logical schema as a conceptual schema. While some constructs are fairly easy to interpret (e.g., a standard foreign key generally is the implementation of a one-to-many relationship type), others pose complex problems due to the use of tricky implementation and optimization techniques.

By construction, these processes are generic, in that they are independent of the DMS, of their data model and of the programming languages used to develop the application.

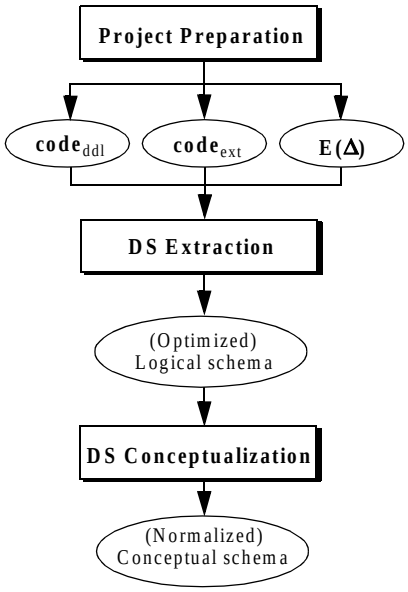


Figure 4-2: General architecture of the reference database reverse engineering methodology.

4.3 The Data Structure Extraction phase

This phase consists in recovering the complete DMS schema, including all the implicit and explicit structures and constraints. True database systems generally supply, in some readable and processable form, a description of this schema (data dictionary contents, DDL texts, etc.). Though essential information may be missing from this schema, the latter is a rich starting point that can be refined through further analysis of the other components of the application

(views, subschemas, screen and report layouts, procedures, fragments of documentation, database content, program execution, etc.). The problem is much more complex for standard files, for which no computerized description of their structure exists in most cases. The analysis of each source program provides a partial view of the file and record structures only. For most real-world applications, this analysis must go well beyond the mere detection of the record structures declared in the programs.

In this methodology, the main processes of *Data structure elicitation* are the following (Figure 4-3).

- **DMS-DDL code analysis**.

  This rather straightforward process consists in analyzing the data structures declaration statements (in the specific DDL) included in the schema scripts and application programs. It produces a first-cut logical schema. Extracting physical specifications from the system data dictionary, such as SQL system catalog, is of the same nature as DDL analysis.

- **Physical integration**

  When more than one DDL source has been processed, the analyst is provided with several, generally different, extracted (and possibly refined) schemas. Let us mention some common situations: base tables and views (RDBMS), DBD and PSB (IMS), schema and subschemas (CODASYL), file structures from all the application programs (standard files), DB schema and COBOL copybooks (source code fragments that are included in the source program at compile time), etc. The final logical schema must include the specifications of all these partial views, through a schema integration process. This process differs from the approaches proposed in the literature on the integration of conceptual views. In particular, we can identify three specific characteristics of physical schema integration.

  1. Each physical schema is a view of a unique and fully identified physical object, namely the legacy database. Consequently, syntactic and semantic conflicts do not represent divergent user views but rather insufficient analysis.
  2. Physical and technical aspects of the data can be used in correspondence heuristics, such as offset and length of data fields.
  3. There may be a large number of such views. For instance, a set of COBOL files serving a portfolio of 1,000 program units will be described by 1,000 partial views. In addition, there is no global schema available for these files. The latter will be recovered by integrating these views.

- **Schema refinement**

  The *Schema refinement* process is a complex task through which various information sources are searched for evidences of implicit or lost constructs. The explicit physical schema obtained so far is enriched with these constructs, thus leading to the *Complete physical schema*. The complexity of the process mainly lies in the variety and in the complexity of the information sources. Indeed, the $\mathbf{code_{ext}}$ part of the system includes,

among others, procedural sections in the application programs, JCL scripts, GUI procedures, screens, forms and reports, triggers and stored procedures. In addition, the non encoded part of the system, i.e. **E(Δ)**, must be analyzed as well because it can provide evidences for lost constructs. This part includes file contents, existing documentation, experimentation, personnel interviews as well as the system environment behavior.

- **Schema cleaning**.

  The specific technical construct such as indexes, clusters and files have been used in refinement heuristics. They are no longer needed, and can be discarded in order to get the complete logical schema.

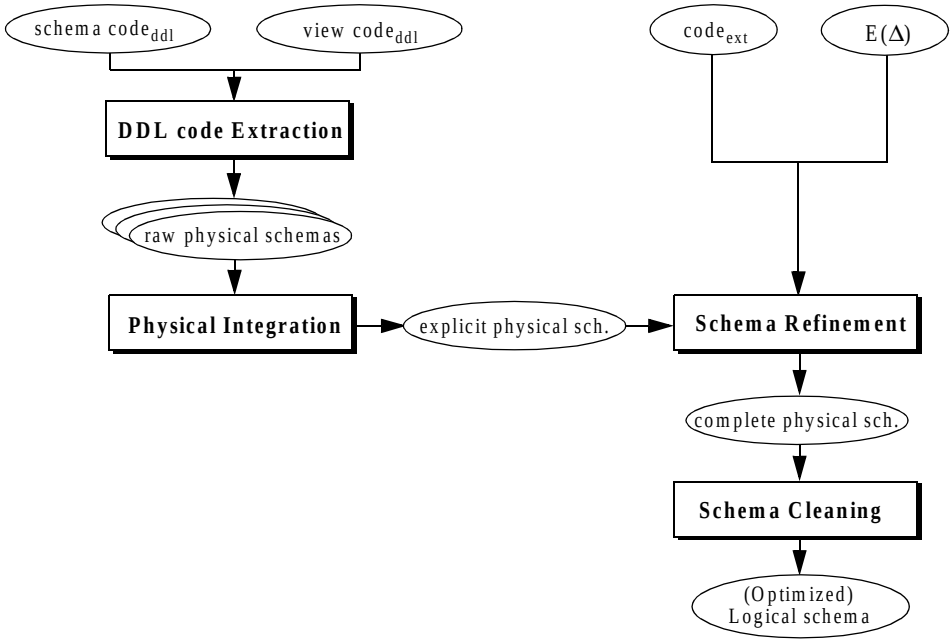


Figure 4-3: General architecture of the *Data Structure Extraction* phase.

It is interesting to note that this schema is the document the programmer must consult to fully understand all the properties of the physical data structures (s)he intends to work on. In some cases, merely recovering this schema is the main objective of the programmer, who can be uninterested by the conceptual schema itself. Hence the importance of identifying this intermediate product, and of defining the independent phases *Extraction* and *Conceptualization*.

4.4 The Data Structure Conceptualization phase

This second major phase addresses the conceptual interpretation of the DMS schema. It consists for instance in detecting and transforming or discarding non-conceptual structures, redundancies, technical optimization and DMS-dependent constructs. Besides the *Preparation phase*, which will not be further developed in this presentation, the Conceptualization phase comprises two main processes, namely *Basic conceptualization* and *Conceptual normalization* (Figure 4-4).

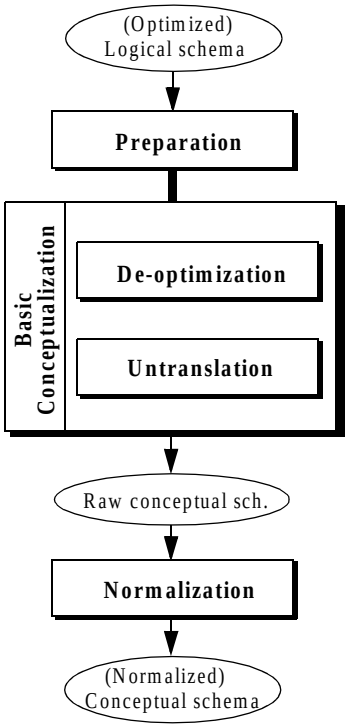


Figure 4-4: General architecture of the *Data Structure Conceptualization* phase.

- **Preparation**.

  The logical schema obtained so far may includes constructs that must be identified and discarded because they convey no semantics. There are two main kinds of such constructs. The *dead data structures* are obsolete, but have been carefully left in the database by the successive programmers. Several hints can help identify them: they are not referred to by any program, they are used by dead sections of programs only, they have no instances, their instances have not be updated for a long time. The *technical data structures* have been introduced as internal constructs for the programs, and are not

intended to model the application domain: program counter, name of the last user, copy of the screen layouts, program savepoints. In addition, this phase carries out cosmetic changes, such as improving the naming conventions and restructuring some awkward parts of the schema.

- **Basic conceptualization**.

  The main objective of this process is to extract all the relevant semantic concepts underlying the logical schema. Two different problems, requiring different reasonings and methods, have to be solved: *schema untranslation* and *schema de-optimization.*

  Schema *untranslation*. The logical schema is the technical translation of conceptual constructs. Through this process, the analyst identifies the traces of such translations, and replaces them with their original conceptual constructs. Though each data model can be assigned its own set of translating (and therefore of untranslating) rules, two facts are worth mentioning. First, several data models can share an important subset of translating rules (e.g. COBOL files and SQL structures). Secondly, translation rules considered as specific to a data model are often used in other data models (e.g., foreign keys in IMS and CODASYL databases). Hence the importance of generic approaches and tools.

  Schema *de-optimization*. The logical schema is searched for traces of constructs designed for optimization purposes. Three main families of optimization techniques should be considered: denormalization, structural redundancy and restructuring.

- **Conceptual normalization**.

  This process restructures the basic conceptual schema in order to give it the desired qualities one expects from any final conceptual schema, such as expressiveness, simplicity, minimality, readability, genericity, extensibility. For instance, some entity types are replaced with relationship types or with attributes, is-a hierarchies are made explicit, names are standardized, etc.

Chapter 5

The data structure extraction process

Abstract

The goal of this phase is to recover the complete DMS schema, including all the implicit and explicit structures and constraints. As explained above, the main problem of the Data Structure Extraction phase is to discover and to make explicit, through the *Refinement* process, the structures and constraints that were either implicitly implemented or merely discarded during the development process. In this section we define the concept of implicit construct, we describe the DDL code extraction process then we analyze the problems and elicitation techniques of implicit constructs. We conclude by the application of these techniques to the recovery of foreign keys.

5.1 Explicit *vs* implicit constructs

An *explicit construct* is a component or a property of a data struture that is declared through a specific DDL statement. An *implicit construct* is a component or a property that holds in the data structure, but that has not been declared explicitly. In general, the DMS is not aware of implicit constructs, though it can contribute to its management (through triggers for instance). The analysis of the DDL statements alone leaves the implicit constructs undetected. The most popular example certainly is that of foreign key, which we will use to explain this point. Let us consider the code of Figure 5-1, in which two tables, linked by a foreign key, are declared. We can say that this foreign key is an explicit construct, insofar as we have used a specific statement to declare it.

```
create table CUSTOMER(C-ID integer primary key,
                      C-DATA char(80))
create table ORDER(   O-ID integer primary key,
                      OWNER integer
                      foreign key(OWNER) references CUSTOMER)
```

Figure 5-1: Example of an explicit foreign key.

Figure 5-2 represents a fragment of an application in which no foreign keys have been declared, but which strongly suggests that column OWNER is expected to behave as a foreign key. If we are convinced that this behaviour must be taken for an absolute rule, then OWNER is an implicit foreign key.

```
create table CUSTOMER(C-ID integer primary key,
                      C-DATA char(80))
create table ORDER(   O-ID integer primary key,
                      OWNER integer)
...
exec SQL select count(*) in :ERR-NBR from ORDER
         where OWNER not in (select C-ID from CUSTOMER)
end SQL
...
if ERR-NBR > 0 then
   display ERR-NBR,'referential constraint violations';
```

Figure 5-2: Example of a (possible) implicit foreign key.

By examining the expressive power of DMSs, compared with that of semantics representation formalisms, and by analyzing how programmers work, we can identify five major sources of implicit constructs.

1. *Structure hiding.*

   Structure hiding concerns a source data structure or constraint S1, which could be imple-

mented in the DMS. It consists in declaring it as another data structure S2 that is more general and less expressive than S1. In COBOL applications for example, a compound/ multivalued field, or a sequence of contiguous fields can be represented as a single-valued atomic field (e.g., a *filler*). In a CODASYL or IMS database, a one-to-many relationship type can be implemented as a many-to-many link, through a record/segment type, or can be implemented by an implicit foreign key. In an SQL-2 database, some referential constraints can be left undeclared by compatibility with older DMSs. The origin of stucture hiding is always a decision of the programmer, who tries to meet requirements such as field reusability, genericity, program conciseness, simplicity, efficiency as well as consistency with legacy components of the application.

2. *Generic expression*.

   Some DMSs offer general purpose functionalities to enforce a large variety of constraints on the data. For instance, current relational DBMSs propose column and table check predicates, views with check option, triggers mechanisms and stored procedures. These powerful techniques can be used to program the validation and the management of complex constraints in a centralized way. The problem is that there is no standard way to cope with these constraints. For instance, constraints such as referential integrity can be encoded in many forms, and their elicitation can prove much more complex than for declared foreign keys.

3. *Non declarative structures.*

   Non declarative structures have a different origin. They are structures or constraints that cannot be declared in the target DMS, and therefore are represented and checked by other means, external to the DMS, such as procedural sections in the application programs or in the user interface. Most often, the checking sections are not centralized, but are distributed and duplicated (frequently in different versions), throughout the application programs. For example, standard files commonly include foreign keys, though current DMS ignore this construct. In the same way, CODASYL DBMSs do not provide explicit declaration of one-to-one relationship types, which often are implemented as (one-to-many) set types + integrity validation procedures.

4. *Environmental properties*.

   In some situations, the environment of the system garantees that the external data to be stored in the database satisfy a given property. Therefore, the developers have found it useless to translate this property in the data structures, or to enforce it through DBMS or programming techniques. Of course, the elicitation of such constraints cannot be based on data structure and program analysis. For example if the content of a sequential file comes from an external source in which uniqueness is garanteed for one of its field, then the database file inherits this property, and an identifier can be asserted accordingly.

5. *Lost specifications*.

   Lost specifications correspond to facts that have been ignored or discarded, intentionally or not, during the development of the system. This phenomenon corresponds to flaws in the system that can translate into corrupted data. However, lost specifications can be

undetected environmental properties, in which case the data generally are valid.

5.2 Explicit construct extraction

This process clearly is the simplest one in DBRE. It consists in associating physical abstractions with each DDL construct. The set of rules is easy to state in most DMSs, provided the target abstract physical model includes a sufficiently rich set of features, such as the one we adopted in Chapter 2. We must be reminded that each DDL, even in the most modern DMSs, includes clauses intended to declare physical concepts (e.g. indexes and clusters), logical concepts (record types, fields and foreign keys) as well as conceptual concepts (identifiers). Distributing the DMS constructs into the standard abstraction levels is sometimes tricky (as in IMS for instance). Table 1 and Table 2 show the main abstraction rules for converting COBOL and SQL-2 code into abstract physical structures. Similar rule sets can be defined for CODASYL, DL/1, TOTAL/IMAGE or OO data structures.

Table 1: Main abstraction rules for COBOL file structures.

COBOL statement	Physical abstraction
`select S assign to P`	define storage S
`record key is F`	define a primary identifier with field *F;* define an access key with field *F.*
`alternate record key is F`	define a secondary identifier with field *F;* define an access key with field *F.*
`alternate record key is F` `with duplicates`	define an access key with field *F.*
`fd S` `01 R`	define record type *R* within storage *S.*
`05 F pic 9(n)`	define numeric field *F* with size *n*, associated with its parent structure (record type or compound field).
`05 F pic X(n)`	define alphanum field *F* with size *n*, associated with its parent structure (record type or compound field).
`05 F1.` `  10 F2 ...`	define compound field *F1*, with components *F2*, etc.
`05 F ... occurs n times`	define multivalued field *F*, with cardinality *n*.

Table 2: Main abstraction rules for relational structures.

SQL statement	Physical abstraction
`create dbspace S ...`	define storage *S*
`create table T (...) in S`	define record type *T* within storage S.
`name numeric(n)`	define numeric field *F* with size *n*.
`name char(n)`	define character field *F* with size *n*.
`... not null`	define the current field as mandatory
`primary key (F)`	define a primary identifier with field(s) *F*
`... unique (F)`	define a secondary identifier with field(s) *F*
`foreign key (F) references T`	define field(s) *F* a foreign key referencing record type *T*.
`create index ...`	define an access key with field(s) *F*.
`create unique index on T(F)`	define a secondary identifier with field(s) *F*; define an access key with field(s) *F*.

Almost all CASE tools propose some kind of DLL extractors for the most popular (generally relational) DBMSs. Some of them are able to extract relational specifications from the system data dictionary as well. Few can cope with non relational structures.

5.3 The Refinement process: information sources and elicitation techniques

Though there exist a fairly large set of potentially implicit constructs, they can all be elicited from the same information sources and through a limited set of common techniques. Figure 5-3 is a summary of the main information sources and of their processing techniques. We will describe them in some detail.

The most common sources of information

Except in small size projects, more than one source will be analyzed to find the data structures and its properties. We will discuss briefly the most common ones.

1. *Generic DMS code fragments.*

   As already discussed, modern database schemas may include SQL code sections that monitor the behaviour of the database. Check/assertion predicates, triggers and stored procedures are the most frequent. They generally express in a concise way the validation of data structures and integrity constraints. They are less easy to analyze since there is

no standard way to code a specific integrity constraint.

2. *Application programs.*

   The way data are used, transformed and managed in the programs brings essential information on the structural properties of these data. For instance, through the analysis of data validation procedures, the analyst can learn what the valid data values are, and therefore what integrity constraints are enforced. This kind of search is called *Usage pattern analysis*.

   Being large and complex information sources, programs require specific analysis techniques and tools. Dataflow analysis, dependency analysis, programming *cliché* analysis and program slicing are some examples of program processing techniques that resort to the domain of *program understanding*. They will be described below.

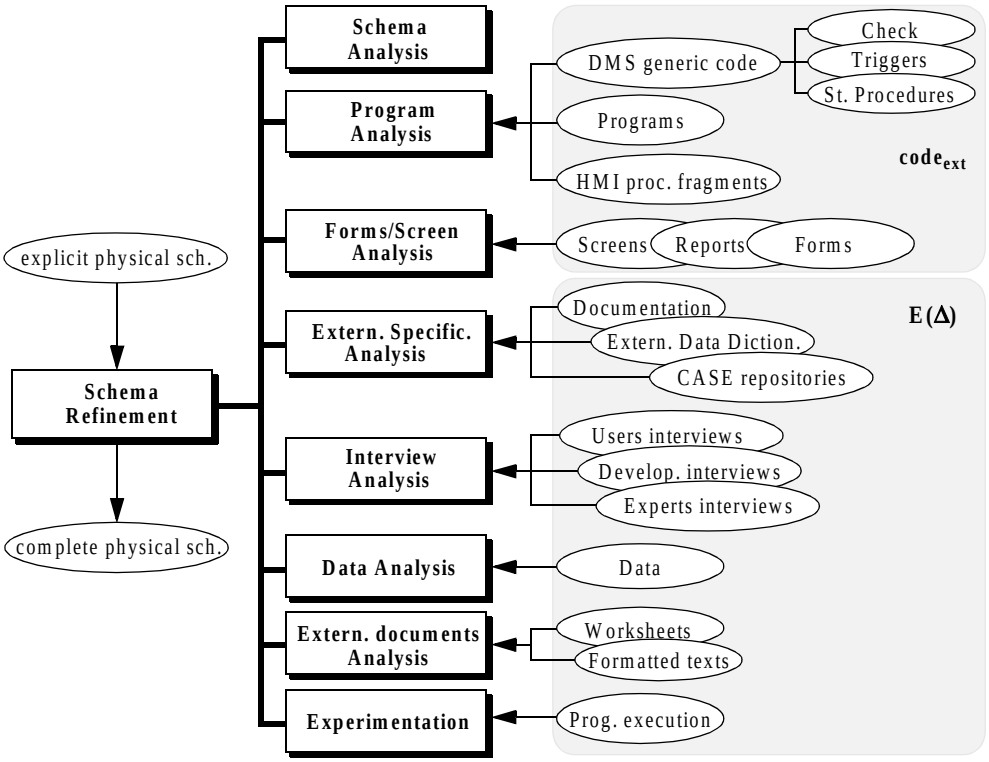


Figure 5-3: Detail of the Schema Refinement process.

3. *HCI procedural fragments.*

   The user-program dialogs generally are monitored by procedures that are triggered by interface events or by database update events. Quite often, these procedures are intended to protect the data against invalid operations, and therefore implement integrity con-

straints validation. *Example*: SQL-Forms in Oracle 5 applications.

4. *Screen/form/report layout.*

   A screen form or a structured report can be considered as derived views of the data. The layout of the output data as well as the labels and comments can bring essential information on the data. Populated screens, forms and reports can be processed through data analysis techniques as well [Choobineh 88; Mfourga 1997].

5. *Current documentation.*

   In some reverse engineering projects, the analyst can rely on some documentation related to the source system. Though these documents often are partial, obsolete and even incorrect, they can bring useful information on the key system components. Of course, the comments that programmers include in the programs can be a rich source of information too, provided their reliability can be assessed.

6. *External data dictionaries and CASE repositories.*

   Third-party or in-house data dictionary systems allow data administrators to record and maintain essential descriptions of the information resources of an organization, including the file and database structures. They can provide informal but very useful description of the data with which one can better grasp their semantics. The main problem with these sources is that they generally have no automatic 2-way link with the databases, and therefore may include incomplete, obsolete or erroneous information. The same can be said of CASE tools, that can record the description of database structures at different abstraction levels. While such tools can generate the database definition code, they generally offer no easy way to propagate direct database structure modifications into these schemas.

7. *Domain knowledge.*

   It is unconceivable to start a reverse engineering project without any knowledge on the application domain. Indeed, being provided with an initial mental model of the objectives and of the main concepts of the application, the analyst can consider the existing system as an implementation of this model. The objective is then to refine and to validate this first-cut model.

   In this context, interviewing regular or past users, developers or domain knowledge experts can be a fruitful source of information, either to build a first domain model, or to validate the model elaborated so far.

8. *Data.*

   The data themselves can exhibit regular patterns, or uniqueness or inclusion properties that provide hints that can be used to confirm or disprove structural hypotheses. The analyst can find hints that suggest the presence of identifiers, foreign keys, field decomposition, optional fields, functional dependencies [Bitton 1989], existence constraints, or that restrict the value domain of a field for instance.

9. *Non-database sources.*

   Small volumes of data can be implemented with general purpose software such as spead-

sheet and word processors. In addition, semi-structured documents are increasingly considered as a source of complex data that also need to be reverse engineered. Indeed, large text databases can be implemented according to representation standard such SGML or HTML that can be considered as special purpose DDL.

10. *Program execution.*

    The dynamic behaviour of a program working on the data gives information on the requirements the data have to meet to be recorded in the files, and on links between stored data. In particular, combined with data analysis, filled-in forms and reports provide a powerful examination means to detect structures and properties of the data.

11. *Technical/physical constructs.*

    There can be some correlation between logical constructs and their technical implementation. For instance, a foreign key is often supported by an index. Therefore, an index can be an evidence that a field could be a foreign key. This source of information as well as the next one can be exploited through *schema analysis* techniques.

12. *Names.*

    Most programmers try to give programming objects meaningful names. Their interpretation can bring some hints about the meaning of the objects, or about their purpose. In addition, this analysis can detect synonyms (several names for the same object) and homonyms (same name for different objects). Fields called *Total_Amount*, *Rebate*, *Shipment_cost* or *Average_Salary* could be derived fields since they suggest values that usually are computed or extracted from reference tables. Program slicing will be the preferred technique to confirm the hypothesis.
    Names can also include important meta-data, such as structural properties (field names *Add-City-Name*, *Add-City-Zipcode* suggest a 3-level hierarchy), data type (*Integer-Level*), unit (*Volume-Tons*), language (*Title-engl*, *Title-germ*).

Some program analysis techniques

Many program analysis techniques have been developed in the softwre engineering domain, e.g., to support program maintenance. Several of these techniques are quite relevant for understanding data structures as well. Here are some of them.

1. *Dataflow analysis.*

   Examining in which variables data values flow in the program can put in light structural or intentional similarities between these variables. For instance, if variable **B**, with structure **Sb** receives its values from variable **A**, with structure **Sa**, and if **Sb** is more precise than **Sa**, then **A** can be given structure **Sb**. The term *flow* must be taken in a broad sense: if two variables belong to the same graph fragment, at some time, and in some determined circumstances, their values can be the same, or one of them can be a direct function of the other. The following table presents some common generating statements:

statement	dataflow graph
`move A to B;`	A ⟶ B
`D := A*B + sqrt(C)`	A ⟶ D B ⟶ D C ⟶ D
`if A = B then ...`	A ⟷ B
`proc P(in X:int; out Y:char);` `P(A,B);`	A ⟶ X Y ⟶ B

More sophisticated, or less strict relations can be used, such as "*if A > B then ...*" and "*C = A + B*". Such patterns do not define equality of values between **A** and **B**, but rather a certain kind of *similarity.* This dependency could be *A and B have compatible value domains*, as considered in problems such as Year 2000 or Dow-Jones 10000.

2. *Dependency analysis.*

   A dependency graph is a generalization of dataflow graphs. It includes a larger variety of inter-variable relations, such as in the following example, where the value of C *procedurally* depends on the value of A.

   :

statements	dependency graph

`if A = 1 then`
`  C := D;`
`endif;`
	A ⟶ C D ⟶ C

3. *Programming cliché analysis.*

   Disciplined programmers carefully use similar standard patterns to solve similar problems. Therefore, once the pattern for a definite problem has been identified, searching the application programs or other kind of procedural fragments for instances of this pattern allows us to locate where problems of this kind are solved [Signore 1994b], [Petit 1994] [Henrard 1998]. For instance, let us consider a complex relational database that includes several thousands of triggers, and whose SQL definition script is, say, more than 3,000 page long. Now, we suppose that we have found that validating a certain kind of integrity constraint IC is carried out through a trigger that requires the size of a query result set to be less than a certain number. Finding all the instances of this validation pattern, and thus finding strong evidences of instances of IC, can be quickly done by searching the database definition script for instances of the pattern of Figure 5-4, or fragments of it. Actual data names have been replaced with pattern variable names (prefixed with @).

4. *Program slicing.*

   This very powerful technique provides extracts from a large program according to defi-

nite criteria [Weiser 1984]. Considering program P, object O of P (e.g., a variable or a record) and a point S in P (e.g., a statement), the program slice Π[P,O,S] of P according to criterion <O,S> is the ordered set of all statements of P that can contribute to the state of O at point S. In other words, executing P and executing Π give O the same state whatever the external condition of the executions.

This technique allows the analysts to reduce the search space when they look for definite information into large programs. Let us consider an example. The most frequent programming pattern used to record data in a database consists in (1) collecting the data, (2) validating the data and (3) writing the data in the database. Examining the validation section should give us essential information on the implicit integrity constraints applicable on the data. Unfortunately, these statements often does not appear as a contiguous code section, but rather are scattered throughout the program, or even in several programs. Locating the point(s) S where the data are recorded (as record type R) in program P is fairly easy (it is a simple programming *cliché* such as "`write R`" or "`rewrite R`" in COBOL). Computing the slice Π[P,R,S] for each point S provides us with a (hopefully) very small subset of P which can be searched for traces of data validation statements. Some examples will be proposed in the case study in Chapter 9.

```
create or replace trigger @TRIG
before insert or update of @COLUMN on @TABLE
for each row
declare
  @EXEPT exception;
  @COUNTER number;
begin
  select count(*) into @COUNTER
  from   @TABLE
  where  @COLUMN1 = :NEW.@COLUMN2;
  if (@COUNTER >= @VAL)
  then
    raise @EXEPT;
  end if;
end;
```

Figure 5-4: A complex text pattern describing a large class of SQL triggers.

About the decision process

The information sources presented above can be used in several ways that define different strategies to apply according to the structure of the problem. Many elicitation processes obey the following pattern.

First, an evidence, obtained by applying an elicitation technique, triggers a question or a suggestion, that will be called a hypothesis (e.g,. *is field F a foreign key?*). If this hypothesis seems valuable, the analyst tries to complete it by using other techniques in such a way that

(s)he can formulate a complete hypothesis (e.g., *is field F a foreign key to record type R*). Then, the analyst will try either to prove the validity of the hypothesis, using hypothesis proving techniques, or to prove that it is false, through hypothesis disproving techniques. In short, for each kind of implicit construct, we can classify the elicitation techniques into:

1. *hypothesis **triggering** techniques*, that put in light a possible implicit construct;
2. *hypothesis **completion** techniques*, that help formulate the complete hypothesis;
3. *hypothesis **proving** techniques*, that tend to increase the confidence in the hypothesis;
4. *hypothesis **disproving** techniques* that tend to decrease the confidence in the hypothesis.

The final word is up to the analyst, who has to close the analysis process, and to decide, according to the set of evidences (s)he has collected so far, whether the hypothesis is converted into a construct or it is refuted. An important consequence of this discussion is that implicit constructs elicitation basically is a decision process, and cannot be a formal, deterministic, process, except in simplistic or well structured applications.

5.4 The Refinement process: representative problems

The variety of implicit constructs can be fairly large, even in small projects, and studying the application of the techniques described above to each of them would deserve a full book of impressive size. The space limit of this chapter suggests just to mention the main implicit structures and constraints that can be found in actual reverse engineering projects of various size and nature. Then, we study one of them, namely implicit foreign keys, in further detail to illustrate the proposed methodology. It is important to keep in mind that this analysis is DMS-independent. Indeed, almost all the patterns that will be discussed have been found in practically all types of databases.

1. *Finding the fine-grained structure of record types and fields.*

   A field, or a full record type, declared as atomic, has an implicit decomposition, or is the concatenation of contiguous independent fields. The problem is to recover the exact structure of this field or of this record type. This pattern is very common in standard file and IMS databases, but it has been found in modern databases as well, for instance in relational tables.

2. *Finding optional (nullable) fields.*

   Most DMS postulate that each field of each record has a value. In general, giving a field no value consists in giving it a special value, to be interpreted as missing or unknown value. Since there is no standard for this trick, it must be discovered through, among others, program and data analysis.

3. *Finding field aggregates.*

   A sequence of seemingly independent fields (ADD-NUMBER, ADD-STREET, ADD-CITY) are originated from a source compound field (ADDRESS) which was decom-

posed. The problem is to rebuild this source compound field. This is a typical situation in relational, RPG, IMS and TOTAL/IMAGE databases that impose flat structures.

4. *Finding multivalued fields.*

   A field, declared as single-valued, appears as the concatenation of the values of a multivalued field. The problem is to detect the repeating structure, and to make the multivalued field explicit. Relational, RPG, IMS and TOTAL/IMAGE databases commonly include such constructs.

5. *Finding multiple field and record structures .*

   The same field, or record structure, can be used as a mere container for various kinds of value. For instance, a CONTACT record type appears to contain records of two different types, namely CUSTOMER and SUPPLIER.

6. *Finding record identifiers.*

   The identifier (or unique key) of a record type is not always declared. Such is the case for sequential files or CODASYL set types for example.

7. *Finding identifiers of multivalued fields.*

   Structured record types often include complex multivalued compound fields. Quite often too, these values have an implicit identifier. For instance, in each CUSTOMER record, there are no two PURCHASE compound values with the same PRODUCT value.

8. *Finding foreign keys.*

   In multi-file applications, there can be inter-file links, represented by foreign keys, i.e., by fields whose values identify records in another file.

9. *Finding sets behind arrays ...*

   Multivalued fields are generally declared as arrays, because the latter is the only construct available in host languages and DMS to store repeating values. Unfortunately, an array is a much more complex construct than a set. Indeed, while a set is made up of an unordered collection of distinct values, an array is a storage arrangement of partially filled, indexed, cells which can accommodate non distinct values. In short, an array basically represents ordered collections of non distinct values with possible *holes* (empty cells). For each array, one must answer three questions: are the values distincts? is the order significant? what do *holes* mean? Clearly, usage pattern and data analysis are the key techniques to get the answers.

10. *Finding functional dependencies.*

    As commonly recognized in the relational database domain, normalization is a recommended property. However, many actual databases include unnormalized structures, generally to get better performance.

11. *Finding existence constraints.*

    Sets of fields and/or roles can be found to be *coexistent*, that is, for each record, they all have a value or all are null. There are other similar constraints, such as *exclusive* (at most one field is not null) and *at least one* (at least one field is not null). These con-

straints can be the only trace of embedded fields aggregates or of subtype implementation (see Figure 2-7).

12. *Finding exact minimum cardinality of fields and relationship types.*

    Multivalued fields declared as arrays, have a maximum size specified by an integer, while the minimum size is not mentioned, and is under the responsibility of the programmer. For instance, field DETAIL has been declared as "occurs 20", and its cardinality has been interpreted as [20-20]. Further analysis has shown that this cardinality actually is [1-20].

13. *Finding exact maximum cardinality of fields and relationship types.*

    The maximum cardinality can be limited to a specific constant due to implementation constraints. Further analysis can show that this limit is artificial, and represents no intrinsic property of the problem. For instance, an attribute cardinality of [0-100] has been proved to be implementation-dependent, and therefore relaxed to [0-N], where N means unlimited.

14. *Finding redundancies.*

    Very often, actual databases include redundancies that are to improve performance. It is essential to detect and express them in order to normalize the schema in further reverse engineering steps.

15. *Finding enumerated value domains.*

    Many fields must draw their values from a limited set of predefined values. It is essential to discover this set.

16. *Finding constraints on value domains.*

    In most DBMS, declared data structures are very poor as far as their value domain is concerned. Quite often, though, strong restriction is enforced on the allowed values.

17. *Finding meaningful names.*

    Some programming discipline, or technical constraints, impose the usage of meaningless names, or of very condensed names whose meaning is unclear. On the contrary, some applications have been developed with no discipline at all, leading to poor and contradictory naming conventions.

Devising a general algorithm organizing the search for these implicit constructs would be an unrealistic attempt. However, we can, without excessive risk, propose a logical hierarchy of goals that should fit most DS Extraction projects. It will be specialized for some DMS in Chapter 7

1. **Refine the record type and field structures** (goal 1). Similarly, find the optional fields (goal 2), the field aggregates (3), the multivalued fields (4) and the multiple field and record structures (goal 5)
2. **Find records identifiers** (goal 6). Similarly, find the identifiers of multivalued fields (goal 7)
3. **Find foreign keys** (goal 8).
4. Find other constraints (goals 9 to 17)

5.5 The Refinement process: application to foreign key elicitation

The problem

The foreign key is a field (or combination thereof) each value of which is used to reference a record in another (or in the same) file. This construct is a major building block in relational databases, but it has been found in practically all kinds of databases, such as IMS and CODASYL databases, where it is used to avoid the burden of explicit relationships, or to compensate the weaknesses of the DBMS. For instance, IMS imposes strong constraints on the number and on the pattern of relationships, while CODASYL DBMSs prohibit cyclic set types. Foreign keys have also been used to ease partitioning CODASYL databases into chunks that need not be on-line simultaneously.

The standard configuration of foreign key can be symbolized by: B.B2 ›› A.A1, where B2 is a single-valued field (or set of fields) of record type B and A1 is the primary identifier of record type A. B.B2 and A.A1 are defined on the same domain. However, practical foreign keys do not always obey the strict recommendations of the relational theory, and richer patterns can be found in actual applications. For instance, we have found a large variety of non standard foreign keys:

- *multivalued foreign keys*: each value of a repeating field is, or includes, a referencing value,
- *secondary foreign keys*: foreign keys that reference secondary identifiers instead of primary ones; these secondary identifiers may even be optional,
- *loosely-matching foreign keys*: the type and length of the foreign key can be different of that of the referenced identifier; for instance, a foreign key of type char(10) can reference an identifier of type char(12), and type numeric(6) can be found to match char(6); in addition the structure can be different: an atomic foreign key can be matched with a compound identifier;
- *alternate foreign keys*: that reference a record of type A, or of type B, or of type C,
- *multi-target foreign keys*: that reference a record of type A, and one of type B, and another one of type C,

- *conditional foreign keys*: that reference a target record only if a condition is met; if the other cases, the value of the "foreign key" is given another interpretation,
- *overlapping foreign keys*: two foreign keys share a common field (e.g. {X,Y} and {Y,Z}),
- *embedded foreign keys*: a foreign key includes another foreign key (e.g. {X,Y} and {Y}),
- *transitive foreign keys*: a foreign key is the combination of two foreigns keys (e.g., C.C3 ›› B.B1; B.B2 ›› A.A1; C.C3 ›› A.A1, where A1 is an identifier of A and B1 an identifier of B).

Let us base the discussion on the schema of Figure 5-5, in which two record types (or tables, or segment types) CUSTOMER and ORDER **may** be linked by a foreign key. We assume that CID is the identifier of CUSTOMER, and that, should a foreign key exist in ORDER, it would reference this identifier (in short, we do not have to elicit the target identifier).

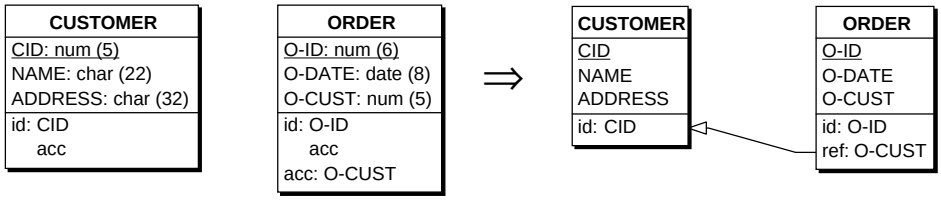


Figure 5-5: Foreign key elicitation - The source and final schemas.

We will examine in which way each of the information sources, techniques and heuristics described in Section 5.3 can be used to elicit foreign key O-CUST, that is, to collect hints and evidences contributing to prove, or disprove that field O-CUST is a foreign key to record type CUSTOMER. Afterwards, we propose a tentative strategy to find implicit foreign keys in relational databases. Its extension to other DMS is quite straightforward.

Though we will discuss the problem of proving that a definite field is a foreign key, it must be noted that this problem may appear in several variants which can be solved by generalizing the techniques that will be examined below. For instance, we could try to find

- all the record types referenced by foreign key O-CUST,
- all the foreign keys in record type ORDER,
- all the foreign keys that reference CUSTOMER.

Dataflow analysis

If a foreign key holds between two record types, then we should find, in some programs, data values *flowing* between variables that represent the foreign key and the target identifier. Considering equality relations only extracted from the program of Figure 5-6, we compute the equality dataflow graph of Figure 5-7.

```
DATA DIVISION.                          PROCEDURE DIVISION.
FILE SECTION.                             ...
FD F-CUSTOMER.                            display "Enter order number "
  01 CUSTOMER.                               with no advancing.
     02 CID pic 9(5).                     accept CID.
     02 NAME pic X(22).                    move 0 to IND.
     02 ADDRESS pic X(32).                 call "SET-FILE" using C, IND.
FD F-ORDER.                               read F-ORDER
  01 ORDER.                                   invalid key go to ERROR-1.
     02 O-ID pic 9(6).                     ...
     02 O-DATE pic 9(8).                   if IND > 0 then
     02 O-CUST pic 9(5).                      move O-CUST of ORDER to C.
                                          ...
WORKING-STORAGE SECTION.                  if C = CID of CUSTOMER then
  01 C pic 9(5).                             read F-CUSTOMER
                                                invalid key go to ERROR-2.
                                          ...
```

Figure 5-6: Excerpts from a program working on CUSTOMER and ORDER (the meaning of this code is irrelevant).

It shows that, at given time points, CUSTOMER.CID and ORDER.O-CUST share the same value. It is reasonable to think that the same property holds in the files themselves, hence the suggestion that field O-CUST could be a foreign key.

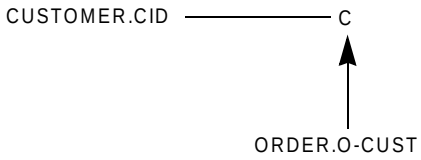


Figure 5-7: Fragment of the equality dataflow graph of the program of Figure 5-6.

Usage pattern analysis

While the dataflow graph only provides us with an abstraction of the relationships between fields, we could be interested by the real thing, i.e., by the procedure that processes records and fields. Let us consider the excerpt of the program of Figure 5-6 that is presented in Figure 5-8(a). A simpler but equivalent version is proposed in (b). The latter exhibits a common *cliché* of file processing programs: reading a record (CUSTOMER) identified by a field value (O-CUST) from another record (ORDER). This is a typical way of navigating from record to record, that is called *procedural join*, by analogy with the relational join of SQL-based DBMS. The probability of O-CUST being a foreign key to CUSTOMER is fairly high.

```
read F-ORDER.
move O-CUST of ORDER to C.
if C = CID of CUSTOMER then
   read F-CUSTOMER
        invalid key go to ERROR-2.
```

(a)

```
read F-ORDER.
move O-CUST of ORDER
     to CID of CUSTOMER.
read F-CUSTOMER
       invalid key go to ERROR-2.
```

(b)

Figure 5-8: (a) excerpts from the program of Figure 5-6. (b) a simplified equivalent version.

Let us consider the following elementary example of foreign key (Figure 5-9).

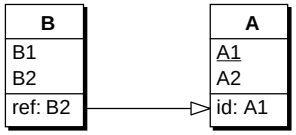


Figure 5-9: An elementary abstract schema including a foreign key.

The main processing patterns that use the instances of this schema are summarized in Figure 5-10. Both procedural (pseudo-code) and predicative (SQL) versions are given.

Function	Procedural pattern
Find the A of a given B	`read A(A1=B.B2);` `if not found then error end-if;`

Find the Bs of a given A	`read-first B(B2=A.A1);`
	`while found do`
	`  process B;`
	`  read-next B(B2=A.A1)`
	`end-while;`

Create a B record	`read A(A1=B.B2);` `if found then create B end-if;`

Delete an A record (*cascade* mode)	`read-first B(B2=A.A1);`
	`while found do`
	`  delete B;`
	`  read-next B(B2=A.A1)`
	`end-while;`
	`delete A;`

Delete an A record (*no action* mode)	`read-first B(B2=A.A1);`
	`is not found do`
	`  delete A;`
	`end-do;`

Function	SQL-like expressions
Find the A of a given B	1)select * from A where A1 in (select B2 from B where ...) 2)select A1,A2 from A,B where A.A1=B.B2
Find the Bs of a given A	1)select * from B where B2 in (select A1 from A where ...) 2)select B1,B2 from A,B where A.A1=B.B2
Create a B record	if exists (select * from A where A1=B.B2) then insert into B values (...)
Delete an A record (*cascade* mode)	delete from B where B2 in (select A1 from A where ...) delete A where ...
Delete an A record (*no action* mode)	if (not exists (select * from B where B2 in (select A1 from A where ...))) then delete A where ... end-if;

Figure 5-10: The main processing patterns related to foreign keys.

Applied on the example of Figure 5-5, some of these patterns could instantiate into the following code sections:

SQL query	CODASYL DML query
select CID, NAME, O-DATE from CUSTOMER, ORDER where CID = O-CUST	move O-CUST of ORDER to CID of CUSTOMER. read CUSTOMER record.

As we can expect, Figure 5-10 gives some popular expressions of the standard processing functions only. Many other variants exist. For instance, SQL offers several ways to code referential integrity implicitly: through check predicates, through SQL triggers, through dialog procedures (e.g., ORACLE SQL-Forms), through stored procedures or through views with check option. In addition, each technique allows the developer to express the constraint validation in his/her own way. Several authors have proposed heuristics to detect foreign keys by usage pattern analysis [Andersson 1994] [Petit 1994] [Signore 1994].

Screen/forms/report layout

Screen forms are used to present data and/or to let users enter them. Frequently, one screen panel includes data from several record types. Typically, an order-entry panel will comprise fields whose contents will be dispatched to ORDER, CUSTOMER, DETAIL and PRODUCT

records. Three kinds of information can be derived from the examination of such forms:

- *Spatial relationships between data fields*. The way the data are located on the screen may suggest implicit relationships.
- *Labels and comments included in the panel*. They bring information on the meaning, the role and the constraints of each screen field.
- *Discarded fields*. If data field O-CUST does not appear on the screen, then its value may appear elsewhere, for instance in screen firld CUST-ID, which has the same type. This may mean that this field designates an information that is given by the context, for instance about the customer of the order.

A screen layout can be examined as a standalone component, as suggested above. It can also be analyzed as source/target data structures of the programs that use it to communicate with their environment. Figure 5-11 shows how screen data are distributed in the data files. An implicit join based on CUSTOMER.CID = ORDER.O-CUST is clearly suggested. Including screen fields in the dataflow graph is another way to make these links explicit

Data reports can be considered both as data structures and as populated views of the persistent data. The first aspect is quite similar to that of screen layout: a report is a hierarchical data structure that makes relationships between data explicit. The second one relates to the data analysis heuristics.

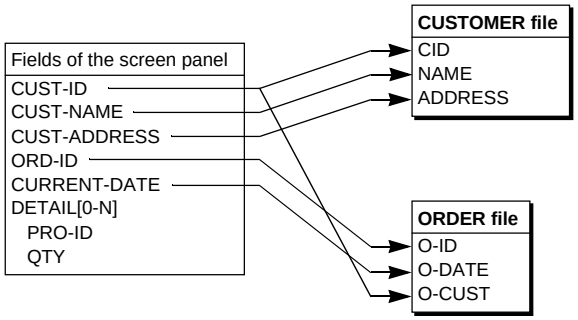


Figure 5-11: Detecting a foreign key in a screen panel.

Current documentation

When it still exists, and when it can be relied on, the documentation is the first information source to use. Normally, file structures, field description, and particularly their roles (such as referencing) should be documented. Some weaker, but probably more up-to-date, information could be found in the system data dictionary. Indeed, most RDBMS allow administrators to add a short comment to each schema object. The programs themselves should include, through *comments*, information on critical components, such as data structures or validation procedures.

Domain knowledge

Everybody knows that *customers place orders*. Obviously, record types CUSTOMER and ORDER should be linked in some way. The question is: *how* ?

Data analysis

If O-CUST is a foreign key to CUSTOMER, then referential integrity should be satisfied, and each of its values must identify a CUSTOMER record. A small program, or the following SQL query (provided the record types are stored into tables) will check this condition by computing the number of violations:

```
select count(*)
from   ORDER
where  O-CUST not in (select CID from CUSTOMER)
```

However the result **n** returned by this query must be interpreted with much caution, because several conclusions can be drawn from it, depending, among others, on the size of the data sample which was analyzed.

Some authors [Petit 1994] have pushed further the analysis of data inclusion properties.

outcome	interpretation
$\mathbf{n} = 0$	1. O-CUST is a FK, 2. statistical accident; tomorrow, the result may be different; O-CUST is not a FK
$0 < \mathbf{n} < \varepsilon$	1. O-CUST is not a FK, 2. O-CUST is a FK, but the query detected data errors, 3. O-CUST is a conditional FK.
$0 << \mathbf{n}$	1. O-CUST is not a FK, 2. O-CUST is a conditional FK.

Program execution

The principle is to analyze the reactions of the program to selected stimuli, for instance in terms of acceptance and rejection of input data and update queries. If the program rejects any tentative data entry concerning an order unless its O-CUST value appears as the CID value of some CUSTOMER record, then we can conclude that the program enforces some kind of inclusion property between these value sets, which can be interpreted as referential integrity. Similarly, if the program refuses to delete a CUSTOMER record because the customer still has pending orders, we can translate this behaviour into the fact that ORDER records depend on this CUSTOMER record.

A running program also populates the screen panels, just like printed reports. New relation-

ships can be detected in this way.

Physical structure

A foreign key is a mechanism that implements links between records, and is the privileged way to represent inter-entity relationships. We can assume with little risk that application programs will navigate among records following these relationships. Therefore, most foreign keys will be supported by such access mechanisms as indexes. *Heuristics*: a field supported by an **index** could be a foreign key, specially when it is not an identifier (most foreign keys implement one-to-many links).

Quite naturally, the candidate field should have the **same domain** of values, i.e., the same type and length, as the identifier of the referenced record type. However, some matching distorsions can be found as far as lengths and even types are concerned. *Heuristics*: the candidate foreign key must match, strictly or loosely, an identifier of the candidate referenced record type.

In some RDBMS (e.g. ORACLE), **clustering** mechanisms are proposed to group in the same pages records that have some kind of logical relationships. Joins based on primary-key/foreign-keys are the most common logical relationship. *Heuristics*: if a physical cluster gathers the records of table A through column A1 and of table B through column B1, and if A1 is an identifier of A, then B1 could be a foreign key referencing A.

In DMS where records can be read in **sorted sequence**, such as in standard file organizations, sorted sets in CODASYL DBMS or in RDBMS (through *order by* clause), it is common practice to interleave records of different types in such a way that they are read in a hierarchical arrangement. Figure 5-12 represents in an abstract way a file comprising ORDER and DETAIL records, sorted on common field ID. Field ID is a global identifier for records of both types. The identifiers (ID) of both types have the same format, and are made of two components. The second component of all ORDER records contains constant value zero, while it is strictly positive in DETAIL records. In this way, each ORDER record is followed by a sequence of DETAIL records that share the same O-ID value. Obviously, DETAIL.ID.O-ID is a foreign key to ORDER, while the *filler* field bears no semantics, and can be ignored. *Heuristics*: in structures such as that described above, the first component of the second record type can be a foreign key to the first record type; in addition, the second component of the first record type can be discarded.

Name analysis

Quite often, the name of a foreign key suggests its function and/or its target. The names can be less informative for multi-component foreign keys. Sometimes, a common prefix may give useful hints. In our example, the name O-CUST includes a significant part of the name of the target record type.

The following rules are among the most common encountered in practice:

- the name suggests the referencing function: CUST-**REF**

- the name suggests the referenced record type: CUSTOMER, CUST
- the name suggests the role of the referenced record type: PLACED-BY, OWNER.
- the name suggests the referenced identifier: CID, CUST-ID, C-CODE

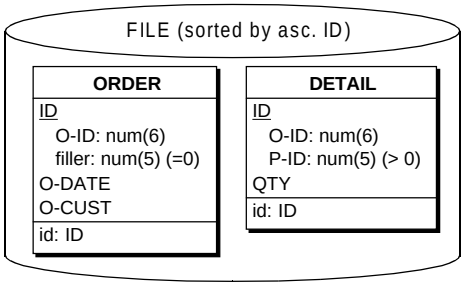


Figure 5-12: Hierarchically sequenced records.

Tentative strategy for foreign key elicitation

Trying to build a general strategy for foreign key elicitation that would be valid in any circumstance would be unrealistic. Indeed, database reverse engineering basically is a loosely structured learning process which varies largely from one project to another. Nevertheless we can sketch the following principles, based on the schema of Figure 5-9, that can apply on relational databases managed by early RDBMS, in which no keys were explicitly declared.

Phase	Heuristics	Short description
Hypothesis Triggering	name analysis domain knowledge	Name of column B.B2 suggests a table, or an external id, or includes keywords such as ref, ... Objects described by B are known to have some relation with those described by A
Hypothesis Comple-tion	name analysis domain knowledge technical constructs	Select table A based on the name of B2 Find a table describing objects which are known to have some relation with those described by B Search the schema for a candidate referenced table and id (with same type and length)
Hypothesis	*B.B2 >> A.A1*	*field B2 of B is a foreign key to identifier A1 of A*

Hypothesis Proving	technical constructs technical constructs dataflow analysis usage pattern usage pattern usage pattern usage pattern usage pattern usage pattern	There is an index on B2 B.2 and A.A1 are in the same cluster B.2 and A.A1 are in the same dataflow graph fragment A.A1 values are used to select B rows with same B2 values B.2 values are used to select A row with same A1 value A B row is stored only if there is a matching A row When an A row is deleted, B rows with B2 values equal to A.A1 are deleted as well There are views based on a join with B.B2 = A.A1 There is a view with check option selecting Bs which match A rows
Hypothesis Disproving	data analysis	Prove that some B.B2 values are not in A.A1 value set

Chapter 6

The data structure conceptualization process

Abstract

DBRE generally is not a linear process. Indeed, trying to interpret logical constructs can suggest new hypotheses that will require the application of elicitation techniques on the operational code or on other sources. So, DBRE really requires a spiral-like approach in which *DS Extraction* and *DS Conceptualization* can be interleaved. For the sake of simplicity, however, we will suppose that we are provided with a logical schema that includes all the constructs that can be elicited from the available sources, and that we will no longer need to go back to DS Extraction activities.

As explained in Section 4.4, the goal of this phase is to extract from the complete logical schema an equivalent normalized conceptual schema. As a side effect, we will often discover *dead data structures* and *technical data structures* that will be discarded, as well as *schema errors* that will be fixed.

Since it has been modeled as the reverse of the forward *Logical Design* phase, the *Conceptualization* phase will be based on transformational techniques as well. We distinguish two major phases, namely *Basic Conceptualization* and *Conceptual Normalization*, that will be described in some detail. The first one is specific to DBRE while the second one is quite similar to the normalization process of the forward *Conceptual analysis* phase (Figure 3-2). The preliminary phase called *Preparation* has been briefly described in Section 4.4 and will ignored in the following.

6.1 Basic conceptualization

This process concentrates on extracting a first cut conceptual schema without worrying about *aesthetical* aspects of the result. Though distinguishing between them may be arbitrary in some situations (some transformations pertain to both), two kinds of reasonings have been identified, namely *untranslation* and *de-optimization.*

The reader will probably be surprised that the processing of some popular constructs will generally be ignored in this section. Such is the case for many-to-many relationship types, N-ary (N > 2) relationship types, attribute entity-types (described later on) and IS-A hierarchies, that are generally considered in most proposals. In fact these problems have been discarded, except when they appear naturally, since they are common to all DMS and there is no general agreement on whether they must be recovered or not. Therefore they will be addressed in the *Conceptual Normalization* process. As an example, most DBRE algorithms[1] automatically interpret an identifying foreign key, that is, a unique key that also is a foreign key, as the trace of an subtype/super-type relation. We prefer to transform it as a mere one-to-one relationship type, which in turn will be interpreted, in the Normalization phase, either as a pure one-to-one relationship type, or as entity type fragmentation or as a subtype/super-type relation.

6.2 Schema Untranslation

At first glance, this process seems to be the most dependent on the DMS model. Indeed a relational schema, a CODASYL schema and a standard file schema, though they express the same conceptual schema, are made up of quite different data structures. They have been produced through different transformation plans (see Figure 3-3 for instance) and therefore should require different reverse *untranslation* rules as well. However, it can be shown that these forward plans use a limited set of common primitive transformations. For instance, the six operators of Figure 2-5 are sufficient to build the major part of the transformation plan of the most popular DMS, and augmenting this toolset with the transformations of Figure 2-6 provides a very rich basis for building optimized schemas. A set of about thirty elementary operators (see Chapter 8) has proved sufficient to define *Logical design* strategies for all past and current DMS, from COBOL standard file systems to OO-DBMS. Since reverse engineering basically is the inverse of forward engineering, a toolbox comprising the inverse of these forward transformations would count no more than one or two dozens of operators.

In fact, in order to make the presentation more attractive, though inevitably redundant, we will discuss untranslation transformations in Chapter 7, devoted to DMS-specific methodologies. However, we will briefly describe five general techniques that will be useful for all the most popular DMS.

1. This presentation should convice the reader that there cannot exist such algorithms, but in simplistic situations. Indeed, DBRE is a highly non-deterministic, decision-based process.

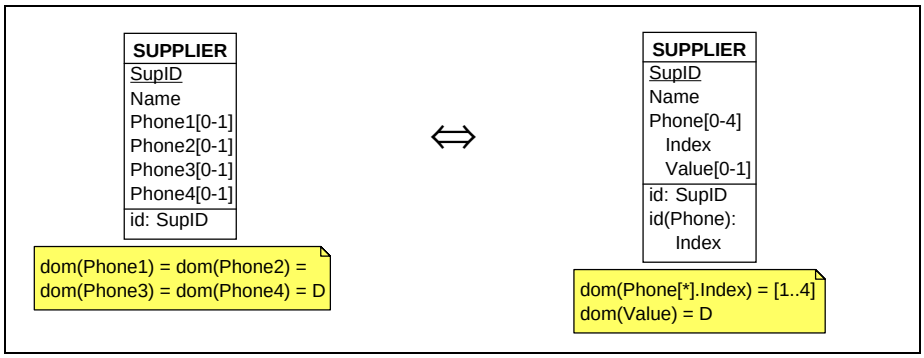


Figure 6-1: Homogeneous serial fields transformed into a multivalued attribute.

Homogeneous serial fields

A group of serial fields is made of a sequence of fields (or attributes, according to the abstraction level of the schema) which present some kind of similarities that is worth being made explicit. Generally, their names suggest some common origin. *Homogeneous* serial fields have the same structure, while their names often includes a common part and a discriminant part. This latter part frequently appears as a set of values belonging to an implicit dimension of the application domain such as months, semesters, years, regions, department, or even mere integers. The untranslation consists in applying a variant of the **Serial-MultAtt** transformation (Figure 6-1) that is more complex but semantics-preserving. The serial fields form an array, which explains the more complex final schema.

Heterogeneous serial fields

Heterogeneous serial fields may have different structures and their names also are made of two parts. However, the discriminant part suggests entity property names. These fields may also form a coexistent group. The suggested untranslation is through the application of the **Aggreg** transformation (Figure 6-2). In some low level languages, such as BASIC in its earlier variants, a program can use a very small number of variables, due to the strong constraints on variable names (1 letter + 1 optional digit). Consequently, programmers used arrays to store records, that is, a list of heterogeneous fields.

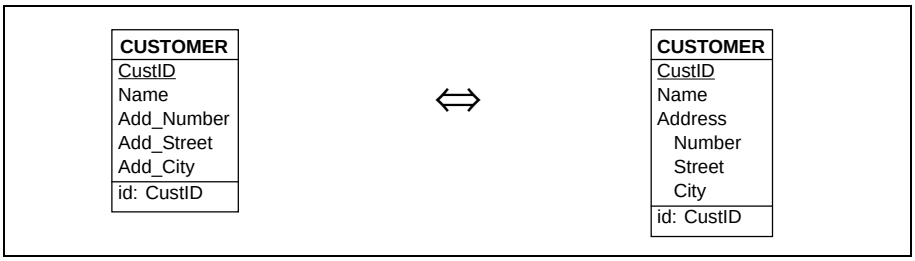


Figure 6-2: Heterogeneous serial fields.

Multivalued compound field

This decomposition technique addresses a pattern that will frequently be found in standard files, but also in any other kind of database, particularly those which do not offer a direct representation of one-to-many relationship type (Figure 6-3). This technique consists in integrating a dependent record type into its master record type as a compound, multivalued field. In this way, it is possible to represent record hierarchies into a single record. Note that this transformation will also be relevant in the *De-optimization* and *Normalization* phases.

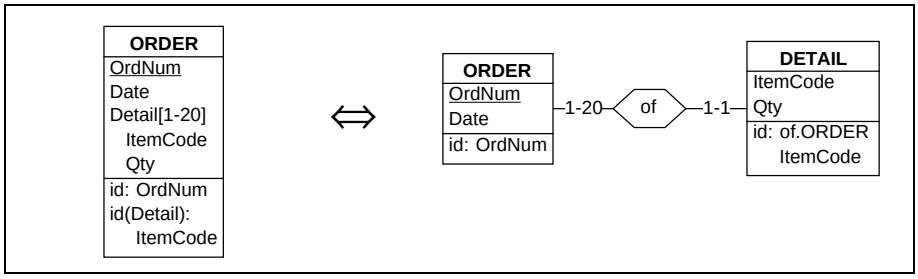


Figure 6-3: Extracting a complex multivalued field as an autonomous entity type.

Hierarchically sequenced records

Some sequences of records or of multivalued attribute values result from the dynastic traversal[2] of an implicit tree. Since all DMS, from sequential files to RDBMS, offer ways to define sequences of records, this pattern can be found in any database. Such a sequence will be replaced with its hierarchical origin (Figure 6-4).

2. I.e., depth-first.

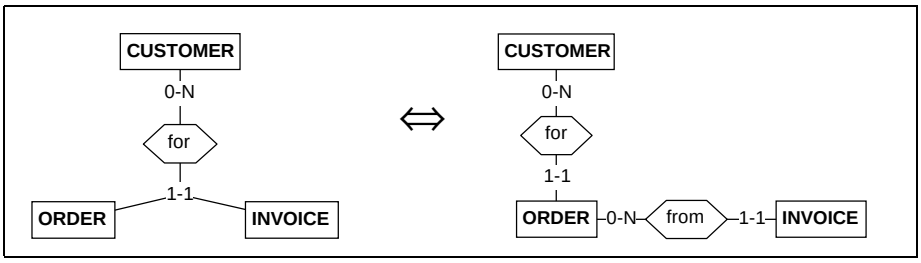


Figure 6-4: A record hierarchy implemented as a record sequence. In the left-side schema, a sequence of interleaved ORDER and INVOICE records depend on each CUSTOMER record. This interleaving structure implements an ORDER-INVOICE hierarchy.

The main problem is to discover how the parent/child relationship is implemented: through a level number, by a node type change, by a specific field value, by a foreign key to the parent node (Figure 5-12), by explicit next/prior pointers, etc. This structure was frequently used in simple file structures and was the basic storage structure of the first versions of IBM's IMS. However, some SQL acrobatic graph coding schemes of the same flavour can be found in recent reference [Celko 1995]. Obviously, reverse engineering such patterns also is for smarties!

Foreign keys

The foreign key is the main inter-entity structuring construct in all the relational or standard file schemas. Surprisingly, it has been found in many schemas implemented in network, shallow, hierarchical, and even OO models. Hence the importance of recovering all the foreign keys in the DS Extraction phase.

The standard foreign key (Figure 6-5) is a group of one or several fields that match the primary identifier of another (or of the same) record type. Its preferred interpretation is as a *many-to-one* relationship type (i.e., transformation **FK-RT**). Variants according to whether the foreign key is optional/mandatory, single/multi-component, identifying or not, single/multi-valued, referencing a primary/candidate identifier, will easily be derived from this standard pattern.

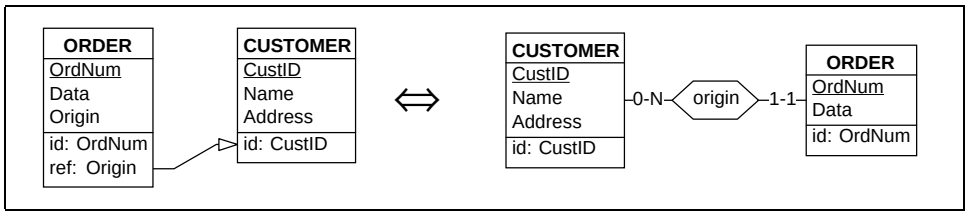


Figure 6-5: Untranslating the standard foreign key.

Unfortunately this transformation does not apply for some non standard patterns. We will ex-

amine some of them and propose a conceptual interpretation[3].

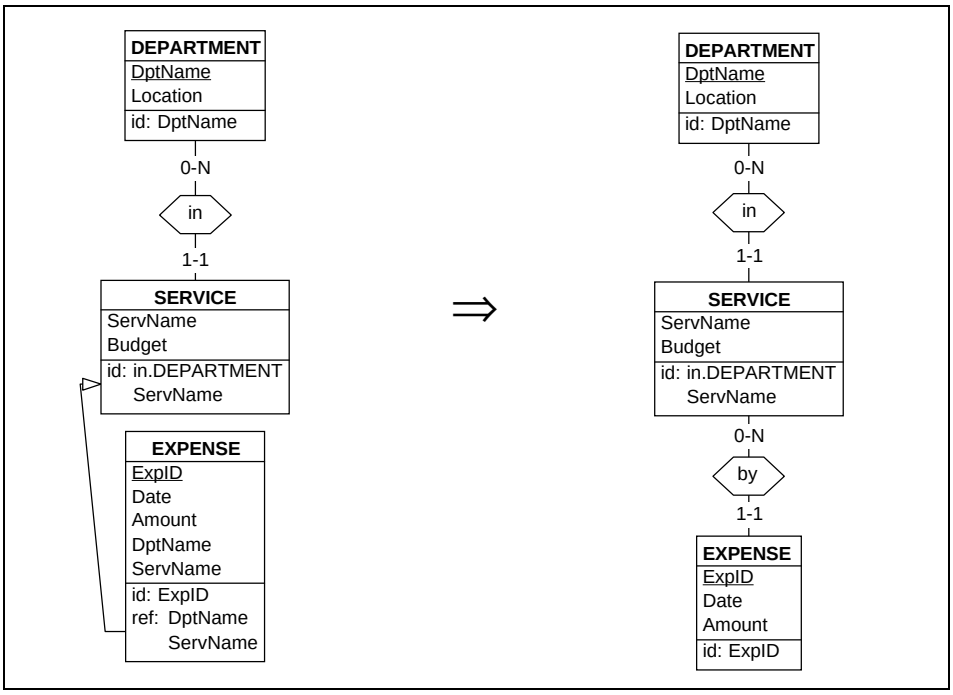


Figure 6-6: Non standard foreign key - Hierarchical foreign key.

- *Hierarchical foreign key.* Surprisingly, foreign keys are very frequent in hierarchical and network database schemas, despite the fact that the DBMSs offer explicit constructs to represent relationship types. If the target entity type has been given an absolute identifier comprising attributes only, these foreign keys are standard. However, if it is identified relatively to one of its parent entity types, the foreign key must reference entities through their hierarchical identifiers (e.g. their concatenated key in IMS) (Figure 6-6). These keys also appear in unfinished conceptual schemas originated in logical schemas from other DMS.
- *Partially reciprocal foreign keys*. This pattern of interleaved foreign keys is a concise and elegant way to represent a bijective (one-to-one) relationship set included into another relationship set (Figure 6-7). When considering its ER equivalent, the source relational schema appears much more concise and free from complex additional integrity constraints. Unfortunately, the price to be paid for this conciseness is that its meaning is far from intuitive.

3. The problem of non standard foreign keys is studied in detail in Chapter **XXXXX**.

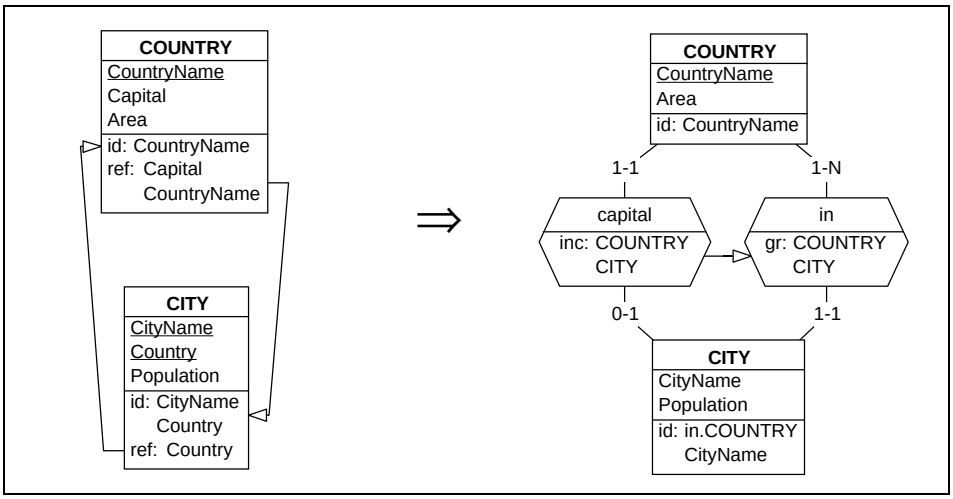


Figure 6-7: Non standard foreign key - Partially reciprocal foreign keys.

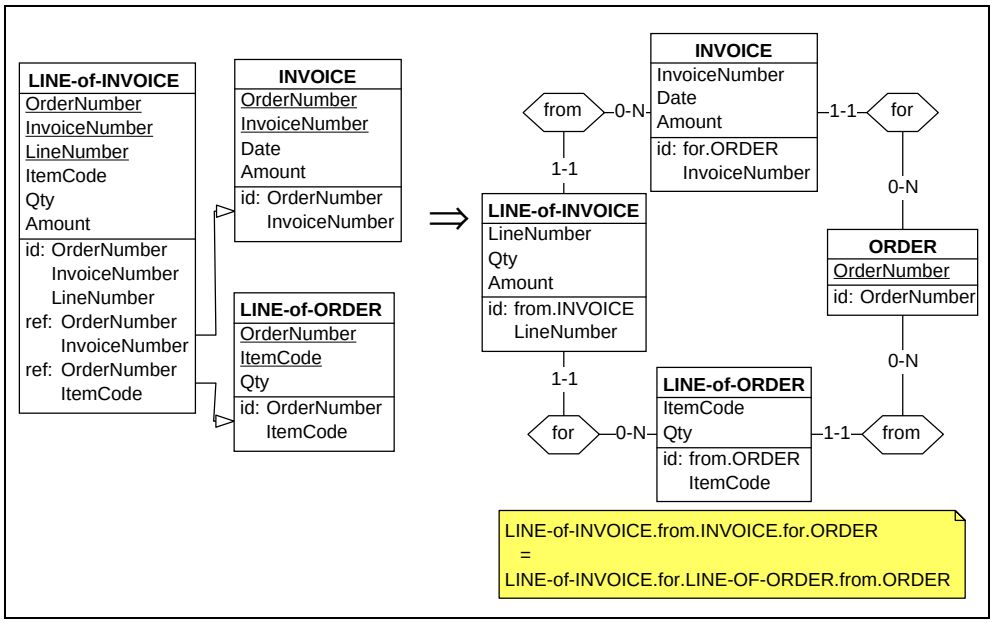


Figure 6-8: Non standard foreign key - Overlapping foreign keys.

- *Overlapping foreign keys.* Two foreign keys overlap if they share one or several columns and if none is a subset of the other one (Figure 6-8). Obviously, we cannot trans-

form one key without destroying the other. The trick is to consider that the common key components form the identifier of a hidden entity type.

- *Partly optional foreign key.* Some, but not all, components of a multiple-component foreign key are optional. First, we define a subtype for which the optional key component is mandatory. All the components of the foreign key become mandatory, so that the latter can be processed as usual (Figure 6-9).

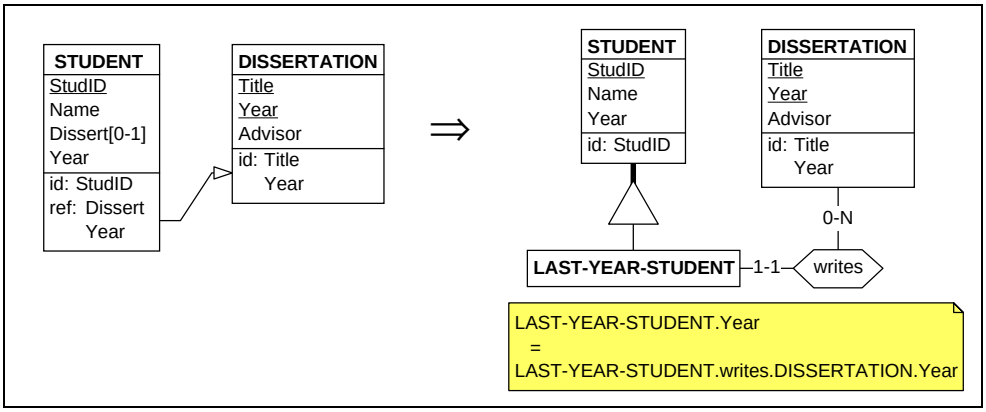


Figure 6-9: Non standard foreign key - Partly optional foreign key.

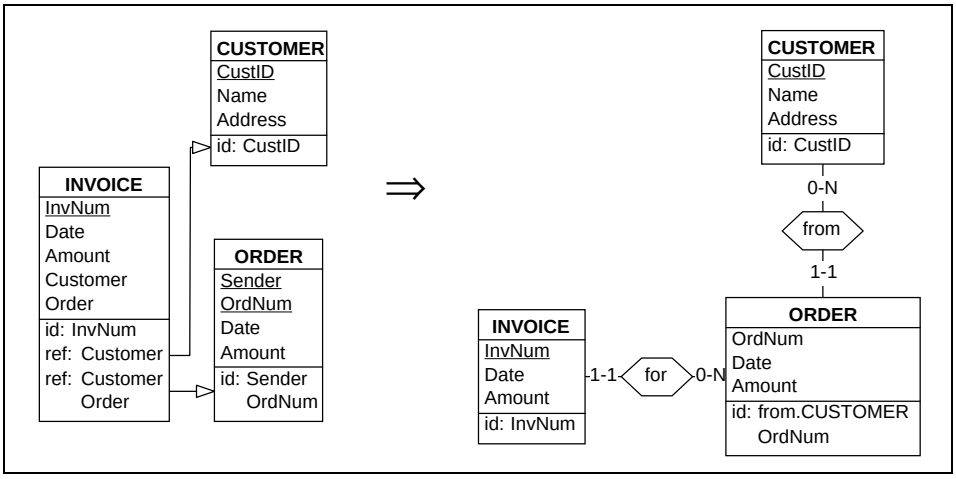


Figure 6-10: Non standard foreign key - Embedded foreign key.

- *Embedded foreign key.* The pattern is as follows: the LHS of foreign key "C.C1 >> A.A1" is a proper subset of the LHS of foreign key "C.{C1,C2} >> B.{B1,B2}". There are several interpretations, but the most frequent one is that the first (embedded) key is

the composition of the second one and of the still undiscovered foreign key "B.B1 >> A.A1". We must first recover the latter, then remove the transitive foreign key. The untranslation is now quite easy (Figure 6-10). Most frequently, this pattern results from the incomplete elicitation of implicit foreign keys.

6.3 Schema *De-optimization*

Both Conceptual and Logical optimization processes will be considered as a whole, since they make use of the same set of transformations, though possibly through different strategies. Let us recall that we have to find traces of four major families of optimization techniques based on schema transformations, namely discarding, structural redundancy, unnormalization and restructuration. They must be precisely understood in order to reverse their effect. In particular, some of them are more specifically fitted for some DMS than for others.

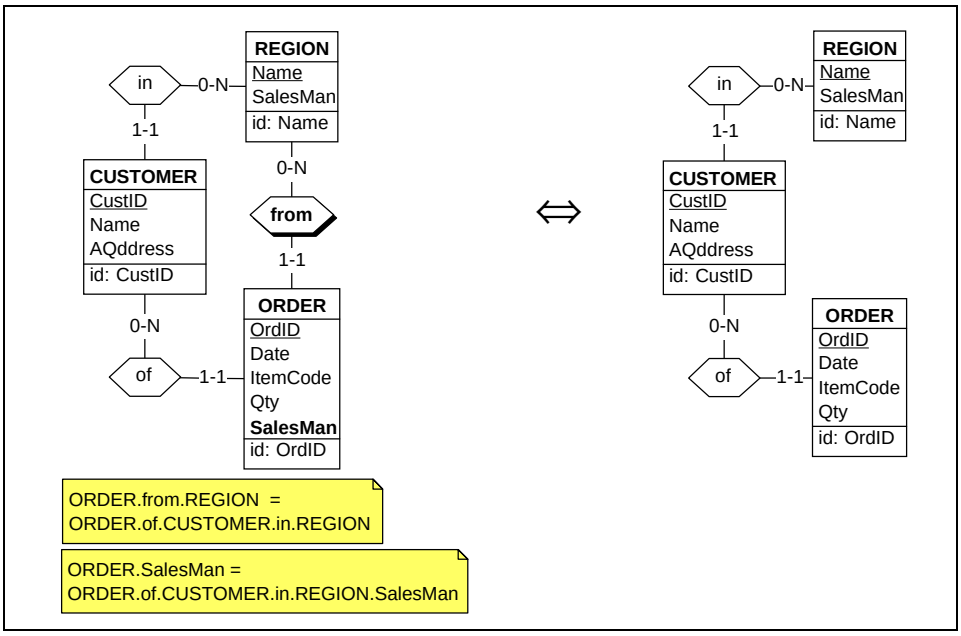


Figure 6-11: Attribute *ORDER.SalesMan* has been recognized (in the DS extraction phase) as duplicate and relationship type *from* as the composition of *of* and *in*. They are removed.

Discarding constructs

This optimization resorts to the lost specification (Δ) phenomenon, and should be addressed

in the DS Extraction phase from sources classified as **E(Δ)** in Figure 5-3.

Structural redundancy

The main problem is to detect the redundancy constraint that states the equivalence or the derivability of the redundant constructs. The expression of such constraints is of the form **C1 = f(C2, C3, ...)**, where **C1** is the designation of the redundant construct. Note that expressions such as **f1(C1, C2, ...) = f2(C3, C4, ...)** generally do not express redundancy, but rather a pure integrity constraint, in which case no constructs can be removed (see Figure 6-8 for instance). Figure 6-11 depicts the elimination of a composed relationship type and of a duplicate attribute.

Normalization redundancy

An unnormalized structure is detected in entity type B by the fact that the determinant of a functional[4] dependency is not an identifier of B . Normalization consists in splitting the entity type by segregating the components of the dependency as illustrated in Figure 6-12. Note that the relationship type should be one-to-many and not one-to-one, otherwise, there would be no redundancy.

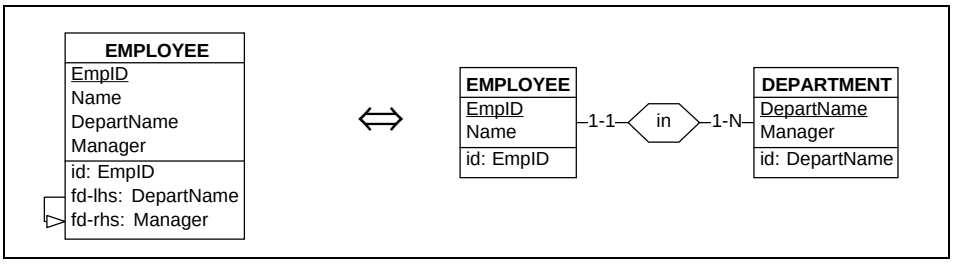


Figure 6-12: Normalization transformation.

Restructuration

We recall that these techniques introduce no redundancy, so that their reversing at this level is not mandatory in most case. For instance, we will find similar reasonings in the *Normalization* phase. We will discuss the reversing of some of the main restructuration techniques.

- *Vertical partitioning optimization*

  Source pattern: entity types E1 and E2 are linked by a one-to-one relationship type and represents complementary properties (attributes or roles) of the same entity type. *Action*: merge E1 and E2.
- *Vertical merging optimization*

4. Considering normal forms based on higher level dependencies is much less useful since they cannot be detected nor even understood by practitioners.

Source pattern: entity type E includes properties related to independent entity types. *Action*: split according to semantic similarities.

- *Horizontal partitioning optimization*

  Source pattern: entity types E1 and E2 have the same properties and represent the same kind of entities. *Action*: remove E1 or E2.

- *Horizontal merging optimization*

  Source pattern: entity type E has unclear semantics that seem to encompass two similar but distinct entity categories. *Action*: either define E1 and E2 as independent entity types, or build an ISA hierarchy in which E1 and E2 are subtypes. *Warning*: this grouping could be the implementation of relationships between the entities (such as in Figure 5-12 for example). In that case a relationship type must be defined between E1 and E2.

- *Complex multivalued attribute*

  We mention this technique since it is frequently found as a way to avoid multiple physical access to dependent record types from the parent record type. It has been presented in Figure 6-3.

- *Hierarchically sequenced records*

  This technique is a popular implementation of relationship types in standard files, but can be found in DBMS offering a direct representation of this construct. In this case, this technique can be considered as an optimization (Figure 6-4).

- *Technical identifier*

  A short primary identifier that bears no semantics, and that has no counterpart in the application domain can be discarded provided the entity type has another identifier.

6.4 Conceptual normalization

Let us first observe that what we call *Normalization* generally does not encompass the relational interpretation of the term. Indeed, relational normalization aims at removing redundancy abnomalies, therefore resorting to de-optimization reasonings.

The goal of the normalization transformations is to improve, if necessary, the expressiveness, the simplicity, the readability and the extendability of the conceptual schema. In particular, we will try to make higher-level semantic constructs (such as ISA relations) explicit. Whether such expressions are desirable is a matter of methodological standard, of local culture and of personal taste. For instance, a design methodology that is based on a binary, functional ER model (e.g., the Bachman's model of the seventies) will accept most of the conceptual schema obtained so far. More powerful models will require the expression of, e.g., ISA relations or N-ary relationship types when relevant. In addition, the final conceptual schema is supposed to be as readable and concise as possible, though these properties basically are subjective. We shall mention some standard transformations that are of interest when refining a conceptual

schema. This list is of course far from complete.

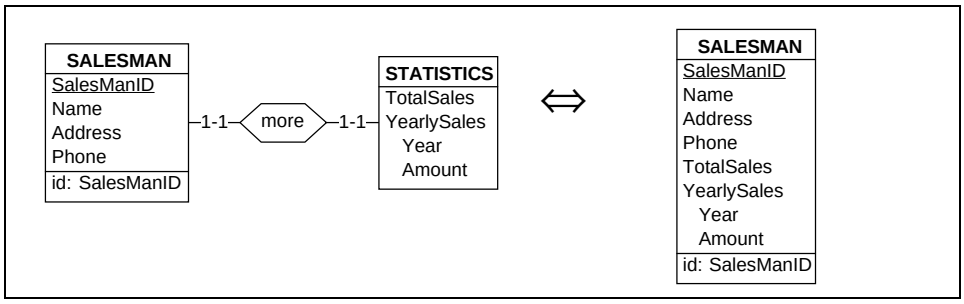


Figure 6-13: Merging two entity types. If the cardinality of *more.SALESMAN* (left) is [0-1], constraint "*coexist: TotalSales, YearlySales*" must be added to SALESMAN (right).

- *Relationship entity type*. By this term, we mean an entity type whose aim obviously is to relate two or more entity types. It will be transformed into a relationship type through transformation **ET-RT**. This technique typically produces many-to-many and N-ary relationship types, as well as relationship types with attributes.
- *Attribute entity type*. Such an entity type has a small number of attributes only, and is linked to one other entity type A through a [1-j] role. All its attributes participate to its identifier. It can be interpreted as nothing more than an attribute of A, possibly multivalued and compound (transformation **ET-Att**.
- *Exact maximum cardinality.* Multivalued attributes and some rel-types are derived from array fields in physical record types. Since the latter construct naturally is limited in size, so are their conceptual interpretation. This limit must be questioned: is it of semantic nature (a person has up to 2 parents) or is it simply inhérited from its physical origin (an order has up to 10 details) ? In the latter case, the limit can be relaxed and set to N.
- *One-to-one relationship type (case 1)*. May express the connection between fragments B1 and B2 of a unique entity type B (vertical partitioning). These fragments can be merged through transformation **Merge** if the resulting schema is clearer (Figure 6-13). However, it may also represent an ISA relation (see case 2 below), or a genuine one-to-one relationship type that must be preserved.
- *Long entity type*. Conversely, an entity type that comprises too many attributes and roles can suggest a decomposition into semantically homogeneous fragments linked by one-to-one relationship types (transformation **Split)**.
- *N-ary relationship type with a [i-1] role*. It can be transformed into binary, one-to-many relationship types through a relational decomposition (Figure 6-14).

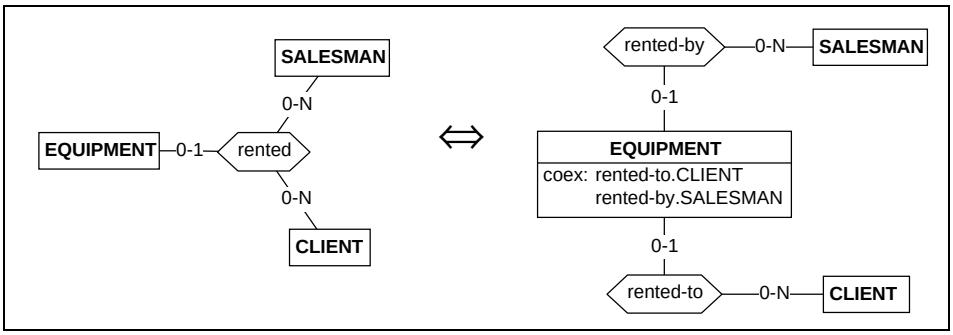


Figure 6-14: Decomposition of a N-ary relationship type through its [i-1] role.

- *Entity types with common attributes and roles*. They can be made the subtypes of a common supertype that inherits the common characteristics (Figure 6-15).

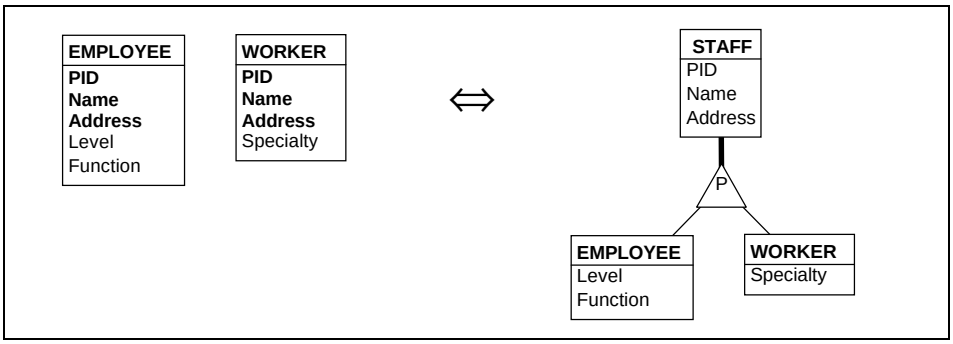


Figure 6-15: Defining a supertype. In fact, the transformation is a bit more complex when constraints, such as identifiers, hold in the source entity types.

- *Groups of coexistent attributes and roles*. Each of them can be extracted as a subtype of the parent entity type (Figure 6-16). An entity type that has one subset of coexistent optional attributes and roles can also be examined for such a transformation. Here, we have applied transformations **Split**, then **RT-ISA**. See also Figure 2-7.

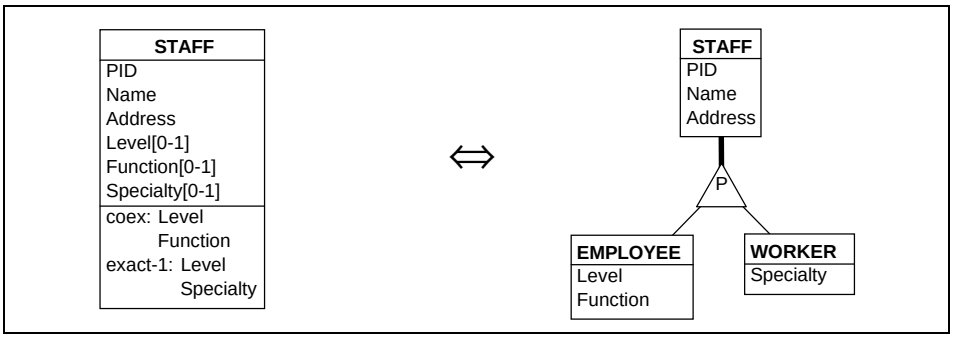


Figure 6-16: Defining subtypes from coexistent subsets of optional attributes/roles.

- *Bunch of one or several one-to-one relationship types (case 2).* If they share a common entity type A, they may express a specialization relation in which A is the supertype. These relationship types are replaced with IS-A relations through transformation **RT-ISA** (Figure 6-17).

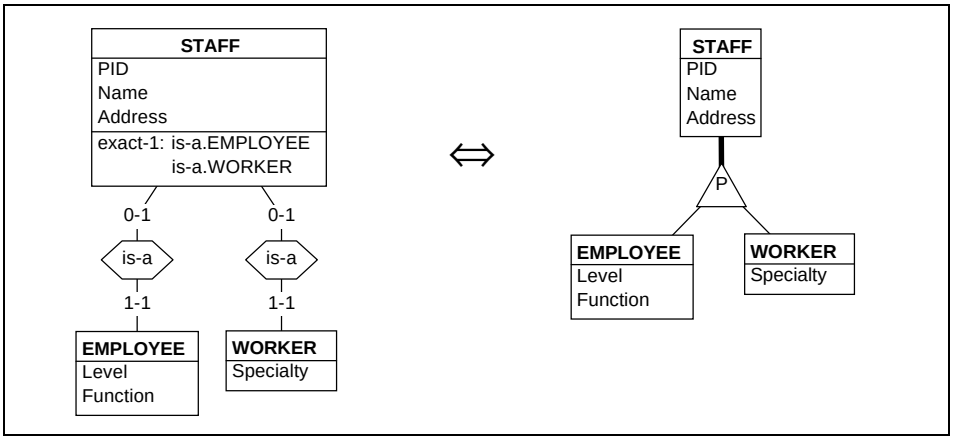


Figure 6-17: Defining IS-A relations from one-to-one relationship types.

Note that the last three techniques simply reverse the three basic expressions of an IS-A hierarchy into the plain ER model as described in [Batini 1992] for instance. They can be specialized in order to make them cover all the situations of total and/or exclusive subtypes. An in-depth analysis of IS-A relations implementation can be found in [Hainaut 1996].

Chapter 7

DMS-oriented DBRE methodologies

Abstract

In sections 5 and 6 we have laid down the basic concepts, reasonings and techniques that are valid for any DBRE project, independently of the DMS. The DMS-specific aspects are very few, mainly comprising the nature of the declared structures such as the DDL, the views mechanisms and the possible data dictionaries. The other processes, such as elicitating implicit constructs, untranslating, de-optimizing and normalizing, are fairly abstract activities that are loosely linked, if ever, with the concerned DMS. In addition, what one could consider as typical relational techniques happen to be frequently used in other DMS models, such OO, IMS or COBOL. However, closing here the discussion on DBRE methodology would be frustrating for the reader, who expects special attention to his/her personal problem context and background. Hence this section, in which we attempt to summarize the main problems that are likely to appear for each of the most popular DMS.

7.1 Standard file reverse enginering

COBOL applications form the most important target of reverse engineering projects. Most of them use simple files to store persistent data. If these data are not too complex, standard files can prove more efficient and far less costly than database technology.

COBOL (or other 3GL) data managers generally offer three kinds of file structures, namely sequential (no identifiers, no access keys), relative (technical numeric identifier and access key) and indexed sequential (any number of identifiers and access keys, each composed of one single-valued field). A file can accommodate records of one or several types. A record type is made of at least one mandatory fields. A field is atomic or compound. It is single-valued or multivalued (array, with a min-max index range). There is no such things as foreign keys nor any constraints but the identifiers.

Variants. Pascal, C, PL/1 and ADA files will be processed in the same way. RPG files are very popular on mid-range systems. They are similar to COBOL files with three main differences: (1) there is a unique file description text, (2) names are up to 6-character long (leading to name interpretation problems) and (3) only atomic, single-valued, mandatory fields are allowed. Reverse engineering old BASIC applications is particularly challenging due to some disturbing characteristics: (1) field names comprise one letter + one optional digit, and generally are meaningless, hence the systematic use of arrays to represent records, (2) a record can be read and written in several parts, through more than one statement and (3) control structures are closer to those of Assembler than of 3GL.

COBOL Data structure extraction

The COBOL language standard provides no global file and record type definitions. Instead, each program specifies on which file(s), on which record type(s), and on which fields it intends to work. Therefore, a tentative global schema must be obtained by merging (*Physical integration* process) the definitions found in a collection of programs. The *DDL code extraction* process has to parse two code sections, namely (1) the File control of the Environment division, that yields file definitions and primary identifiers and (2) the File section of the Data division that provides record types structure (see Table 1). In disciplined environments, copybooks (see 9.4.3, *Physical integration*) can be considered as a primitive form of data dictionary. A COBOL-specific *Schema Refinement* phase could be as follows[1]:

1. The most important constructs are in bold face.

1. **Refine the record type and field structure**
2. Find the implicit optional, compound or multivalued fields
3. Find the multiple field and record structures (rename and redefine)
4. **Find the missing records identifiers**, particularly in sequential and relative files
5. Find the identifiers of complex multivalued fields
6. Interpret arrays as sets or lists
7. **Find foreign keys**, including those embedded into multivalued fields.
8. Find existence constraints among optional fields
9. Refine min-max cardinalities of multivalued fields
10. Identify redundancies
11. Find enumerated value domains and constraints on value domains

This process has to be iterated since the discovery of a construct can suggest the existence of another one.

COBOL Basic conceptualization

The COBOL data model imposes few constraints on field structures. The most important one concerns multivalued fields, which can be represented through *arrays* only. However, some programmers adopt relational-oriented representations such as concatenation (**MultAtt-Single**) or instantiation (**MultATT-Serial**) that can be processed as illustrated in Figure 6-1. In addition, optional fields can be represented as multivalued fields with max cardinality of 1 (`... occurs 1 depending of ...`). Compound fields will be decomposed if they were artificially aggregated due to the fact that an index can be defined on one field only. Record fields can overlap (through `rename` and `redefine` statements), leading to multiple definition structures that must be clearly understood. Three cases must be considered:

1. A field definition F2 is compatible (e.g., has same total length) with, but more detailed than field definition F1. *Possible interpretation*: F2 is a more detailed view of F1; keep F2 and discard F1.
2. A field definition F2 is incompatible with field definition F1. *Possible interpretation*: each definition represents the specific properties of a different category of records; define as many subtypes as there are definitions.
3. A record definition R2 is incompatible with record definition R1. *Possible interpretation*: there are two different record types, possibly linked through an implicit relationship type (see Figure 5-12 and Figure 6-4).

The absence of explicit rel-type representation is a more challenging constraint. The most popular representation is through foreign keys, that can be multivalued (Figure 7-1) and non standard. Note that this technique requires that the foreign key is a the top level field. Should it be a component of a compound field, the compound attribute should be processed first (by transformations **Disagg** or **Att-ET/inst** for instance). The foreign key is reversed through transformation **FK-RT**.

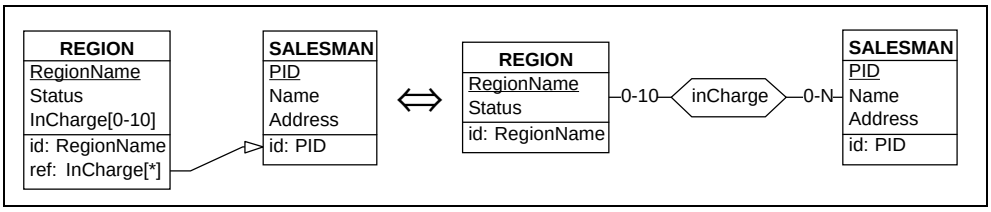


Figure 7-1: Representation of a foreign key by a relationship type. Due to the cardinality of the foreign key, the latter is many-to-many.

Another technique consists in implementing a one-to-many relationship type R between entity types A and B by integrating B entities as instances of a multivalued, compound, attribute of A. Recovering the origin constructs R and B can be done by applying transformation **Att-Et/inst** on the complex attribute, promoting it to an entity type, as illustrated in Figure 6-3.

Finally, a one-to-many rel-type can be represented by a sorted multi-record-type, sequential or indexed file (Figure 5-12).

7.2 Hierarchical database reverse enginering

IMS has long been (since 1968) the main database engine on IMB large systems, progressively replaced with DB2 for more than a decade and a half. Many current mission critical, high performance, batch and transactional applications still are based on IMS databases. Originated in the mid-sixties as a tape-based hierarchical data manager, it has grown, rather inelegantly, to a full-fledge database and data communication system. The data model is awkward, plagued by numerous ill-integrated incremental add-ons and perplexing limitations that makes it a real delight for DBRE researchers, but a bit less appreciated by practitioners. Much more than the CODASYL proposals, IMS was the main advocate for the relational revolution in the late seventies.

The IMS (improperly called DL/1) model structures a schema as a collection of segment types (record types), linked by one-to-many relationship types, that fall into two classes, physical and logical [Elmasri 1997]. The *one* side of a relationship type is a physical or logical parent while the *many* side is a physical or logical child. Ignoring logical relationship types, the schema reduces to a forest, i.e. a collection of trees (or physical DB's). The root of each tree is called a root segment type. Each root can have one identifier, that is an access key and can be a sort key as well. It consists of one field. Each relationship type defines an access path, from the parent to the child only. A child segment type can have an identifier made of its parent + one local field. This identifier is not an access key. Fields are mandatory, single-valued and atomic. However, compound attributes can be simulated by defining overlapping attributes through common physical positions. Due to the weakness of field structures, many segment types are defined with a long, anonymous data field that is redefined

through copybooks or program local variables.

Most IMS databases are built with the latter physical constructs. However, two additional features can be used, namely logical relationship types and secondary indexes. A *logical relationship type* represents an access path from a (logical) child segment type to a parent one. A logical relationship type can be defined between any two segment types provided some exotic constraints are satisfied: a segment type can have only one logical parent, a logical child must be a physical child (i.e., not a physical root), a logical child cannot be a logical parent, the physical parent of a logical child cannot be a logical child, etc. When bi-directional access paths are needed, IMS proposes to define two, inverse, logical relationship type structures (the pairing technique). A *secondary index* is an access key based on any field hierarchy of the database, whatever their segment type. Surprisingly enough, logical relationship types and secondary indexes are considered in the IMS world as intimidating constructs which are difficult and dangerous to use. Even rather recent references [Geller 1989], though insisting on their harmlessness, suggest to avoid them whenever possible, for instance by replacing relationship types by foreign keys controlled by the application programs.

This description delineates clearly the main problems that will appear when translating an conceptual schema into an IMS structure: ISA relations, compound and multivalued attributes, entity types with several identifiers, one-to-one, many-to-many or cyclic relationship types, circuits, entity types with more than two parents, complex identifiers.

IMS Data structure extraction

The main textual sources of **explicit data structure** definition are:

- the physical Database Description, that defines each physical hierarchy, i.e., the segment types, their fields, their physical relationships and their logical relationships, if any; duplicate structures resulting from segment pairing can be resolved easily since they are explicitly declared. The redundant logical children can be merged in a preliminary step.
- the logical Database Descriptions, the Secondary Index specification and the Program Communication blocks, that each define a sort of view of the database intended to define access paths to the data;
- host language Copybooks, that redefine segment types.

Merging DL/1 definitions and Copybooks definitions can be straighforward when they are compatible, but will require more care if they are conflicting, in which case they may suggest supertype/subtype structures.

The following list of operations could form the basis for an IMS-specific *Schema Refinement* phase:

1. **Refine the segment type and field structure**
2. Find the implicit optional, compound or multivalued fields
3. Find the multiple field and record structures (rename and redefine)
4. **Find the missing segment identifiers**, particularly in non-root segment types that depend on two parents
5. Interpret arrays as sets or lists
6. **Find the implicit foreign keys**
7. Find existence constraints among optional fields
8. **Refine min-max cardinalities of relationship types** (many of them actually are one-to-one)
9. Refine min-max cardinalities of multivalued fields
10. Identify redundancies such as duplicated segment types
11. Find enumerated value domains and constraints on value domains

IMS Basic conceptualization

Many of the conceptualization problems can be considered to be relevant to conceptual normalization (e.g., recovering many-to-many or cyclic relationship types). We shall concentrate on recovering non-compliant one-to-many relationship types. We know that they have been transformed, most often, into foreign keys (manually controlled) as in relational schemas, by merging their entity types (producing a possibly unnormalized structure), or into relationship entity types. Recovering the source relationship type from application of latter technique is described in Figure 7-6. However, to make the process clearer, processing a typical IMS substructure is depicted in Figure 7-2, where entity types F and G have been considered as one-to-many rel-type representations. Once again, the main difficulty is to detect the one-to-one cardinality of R5 and R7. R6 and R8 (or R5 and R7) can be implemented trough foreign keys, possibly supported by secondary indexes.

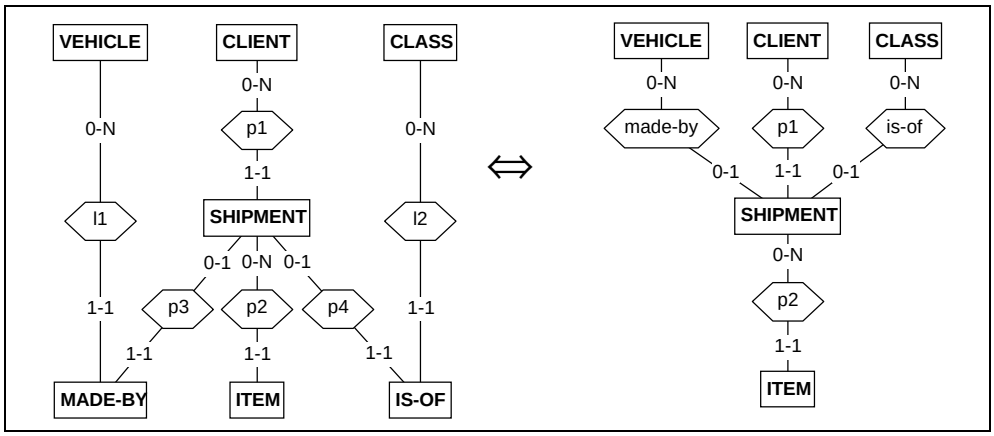


Figure 7-2: Recovering one-to-many rel-types in a typical hierarchical schema.

Due to the weakness of the hierarchical model, many schema include redundant segment types and redundant fields. Identifying and discarding them is the main task of the de-optimization phase.

IMS structure understanding has gained a new interest with the emergence of XML as a way to organize permanent data. Indeed, an XML database structure is made up of trees of elements that can be linked with references (IDREF in XML or REF in XML schemas). Therefore, all the heuristics that IMS database designers used to apply still are valid for structuring XML databases.

7.3 Network database reverse enginering

The name *network* generally designates some variant of the CODASYL DBTG data model recommendations. About twenty of them have been implemented, ranging from high-end mainframe DBMS (IDMS, IDS-2, UDS) to small system DBMS (SIBAS, MDBS). Despite their qualities, such as the richness of their data model, their tight interface with COBOL (both come from the CODASYL group) and their performance, they have been progressively replaced by RDBMS. However, many CODASYL-based legacy systems still are active, probably for many years.

Among the logical models considered in this chapter, this one is the closest to the Entity-relationship model. A schema is made of record types and set types (one-to-many, non cyclic, relationship types). A set type is made of one *owner* record type and one or several *member* record types. A record type can have an arbitrary number (including none) of fields. A field can be optional/mandatory, single-valued/multivalued (array) and atomic/compound. A field-only identifier can (but must not) be defined on each record type. An arbitrary number of identifiers can be declared among the members of a set type (each represents an identifier made of one role and one/several fields).

Each schema includes the *SYSTEM* record type that has one and only one instance. A *system set type* is a set type whose owner is *SYSTEM* and whose member is a user-defined record type. It is called *singular* since there only one instance of this type. A system set type with member M is used for several purposes: to define a subset of M records, to define a secondary all-field identifier for M, to define a secondary field-only indexes on M and to define sorted access paths to M records. Since these constructs are very frequent, it is important to identify the goal of each system set type.

CODASYL Data structure extraction

The main source of explicit structures is the schema description coded in *Schema DDL*. Quite frequently, subschemas (i.e., users views) express more detailed data structures, and are worth being analyzed. The physical DDL (DMCL) specifications generally are useless and can be ignored. Copybook are frequently used, and provide the same information as users

views.

The main implicit constructs to search for are the field decomposition, the foreign keys, the exact cardinalities of roles and fields and the complex record type identifiers. Redundancies are less frequent than in hierarchical schemas, but are worth being paid some attention. We can suggest the following searching script, that will apply in most cases.

1. **Refine the record type and field structure**
2. Find the implicit optional, compound or multivalued fields
3. Find the multiple field and record structures (rename and redefine)
4. Identify clearly the role of each SYSTEM set type
5. **Find the missing record identifiers**, particularly those which include more than role as well as the secondary field-only identifiers
6. Interpret arrays as sets or lists
7. **Find the implicit foreign keys**
8. Find existence constraints among optional fields
9. **Refine min-max cardinalities of relationship types** (some of them actually are one-to-one)
10. Refine min-max cardinalities of multivalued fields
11. Identify redundancies
12. Find enumerated value domains and constraints on value domains

CODASYL Basic conceptualization

Due to the richness of the CODASYL model, there are less structural problems to be solved when translating a conceptual schema into a logical schema than for any other DMS model. As far as conceptual structures are concerned, the main restrictions apply on relationship types (one-to-many and non-cyclic) and on identifiers (one absolute id through *location mode calc*; one relative id per relationship type through *duplicates not* in the member clause). Therefore, non-binary relationship types, many-to-many relationship types, one-to-one relationship types, cyclic relationship types, secondary field-only identifiers, identifiers with more than one role, have to be transformed.

Recovering non-binary and many-to-many relationship types will be considered as the target of conceptual normalization (**ET-RT**), and will be ignored in this section. A one-to-one relationship type is implemented either by a one-to-many relationship type limited by a cardinality constraint, or by a foreign key. Evidence of the first technique will be found through procedural code analysis and data analysis. Processing the second technique is similar to the situation of COBOL and relational schemas. A cyclic relationship type can be represented by an entity type and two one-to-many or one-to-one relationship types. Recovering such a relationship type falls in the conceptual restructuring techniques. It can also be represented by a foreign key, as for COBOL and relational models (Figure 6-5).

An record type with K field-only identifiers will be inserted into (K-1) system set types, i.e., relationship types whose (0-N) role is played by the SYSTEM entity type. Each of the (K-1) secondary identifiers is declared local within one of each such SYSTEM relationship type.

The origin identifiers are recovered by discarding the SYSTEM component (Figure 7-3).

Another implementation technique consists in extracting the attributes of the identifier to transform them into a record type, linked to the main record type through a one-to-one relationship type. This construct can be detected as an *attribute entity type* in the Conceptual Normalization process.

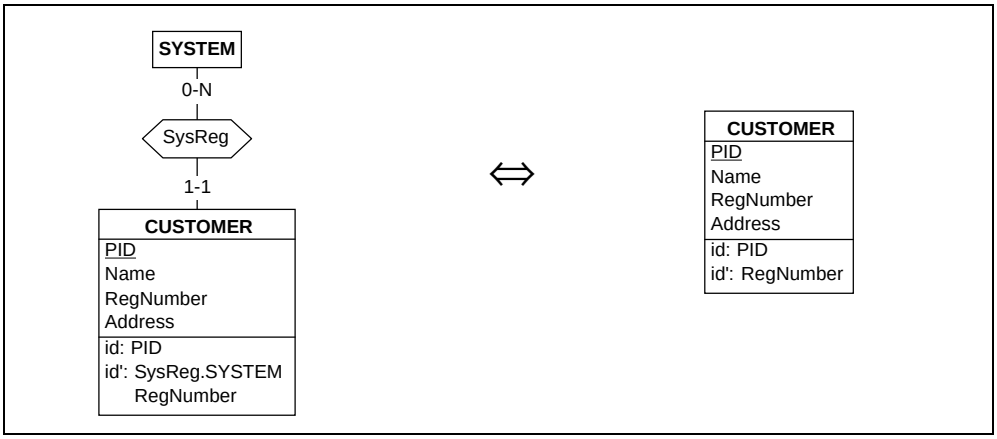


Figure 7-3: Recovering a secondary attribute-only identifier.

A complex identifier that includes more than one role component cannot be expressed as such. Either this identifier is discarded from the schema, and processed by procedural sections, or all the role components but one are replaced by foreign keys.

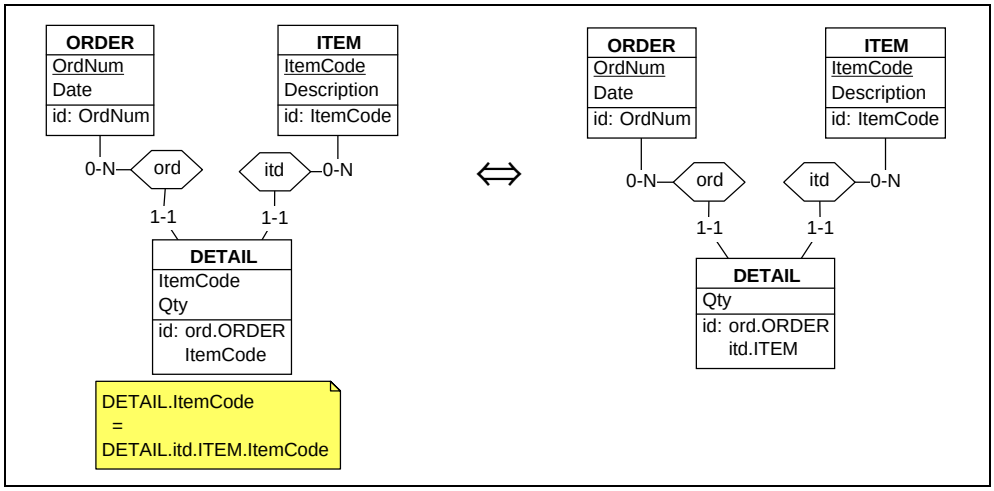


Figure 7-4: Recovering a complex identifier (with more than one role component).

The latter technique can be reversed as proposed in Figure 7-4. The schema may keep the source rel-type, according to the principles of non-information bearing sets as proposed in the 70's [Metaxides 1975]. In such situations, some DBMS offer a trick (an option of the *set selection* clause) through which the referential constraint is automatically maintained.

The optimization techniques are less frequent in CODASYL schemas. However, it would be wise to search them for constructs such as:

- composed set types (Figure 6-11)
- in a set type, member attributes that are copied from owner attributes (Figure 6-11).
- hierarchically sequenced records (Figure 6-4).

7.4 Shallow database reverse enginering

The TOTAL DBMS (CINCOM), and its clone IMAGE (HP), have been intensively used on small to medium systems to manage complex data in the seventies and eighties. The model and the API both are simple as compared with those of IMS. The implementation is light and efficient, and the database applications can run in small machines. Recovery and concurrency management is rather primitive.

These DBMS propose very similar logical models (IMAGE offers additional secondary index structures) that are generally classified as network [Tsichritsis 1977]. However, this common model seems to fit also into the hierarchical model philosophy as far as design techniques are concerned. It offers two kinds of record types, namely the master record types (*master* data set), and the detail record types (*variable entry* data set). In addition one-to-many relationship types can be defined between record types; each relationship type defines access-paths from the master records to detail records. A master has single-valued, mandatory and atomic fields, one of them being its identifier and access key; it can be origin (one side) of a relationship type. A detail is the target (many side) of at least one relationship type. A detail record is the target of at least one relationship type instance (the others can be optional). Among its fields, there is a copy of the identifier value of each of its parent master. These copies behave like redundant foreign keys that allow accessing the parent records. A detail has no identifier. A TOTAL/IMAGE schema is a two-level hierarchy - sometimes called a *shallow* structure - in which level 1 comprises masters only while level 2 is made of details only.

TOTAL/IMAGE Data structure extraction

The explicit data structures are extracted from the DDL source texts. All the other sources mentioned in Section 5.3 will be used for recovering implicit constructs. The reasonings are close to those applicable to CODASYL and IMS as far as relationship types are concerned, including the elicitation of cardinalities and implicit identifiers. TOTAL allowing flat field structures only, field refinement involves techniques which are used when processing relational databases. Due to the similarity with these DMS, we do not propose a specific script

for the *Schema Refinement* phase.

TOTAL/IMAGE Basic conceptualization

In this model, the problems that occur when translating ER to TOTAL/IMAGE are numerous: expressing complex attributes, non-functional rel-types, one-to-one rel-types, cyclic rel-types, relationship type hierarchies with more than two levels and non-hierarchical schemas, entity types with more than one identifier or with secondary access keys, just to mention the most important. Systematic translation of some of these constructs has been proposed in [Hainaut 1981].

In reverse engineering TOTAL/IMAGE schemas, the redundant foreign keys are detected without problem since they are explicitly declared; they can be discarded without loss. Compound attributes are most often processed in the same way as in relational schemas. A multivalued attribute can be detected either in the same way as in relational schemas or as a single-attribute, single-parent, detail entity type (Figure 7-5).

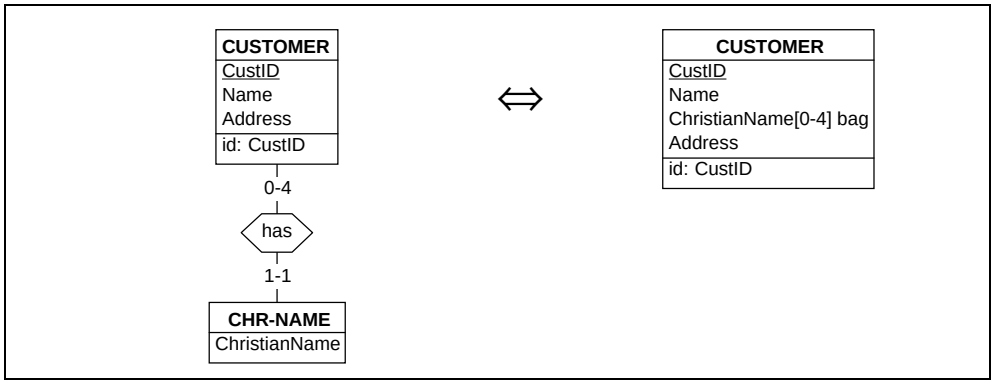


Figure 7-5: Recovering a multivalued attribute. Since entity type CHR-NAME has no identifier, it has been transformed into a bag attribute. Further analysis could lead to the discovery of some identifying properties that would turn it into a set attribute.

One-to-one relationship types can be processed as in CODASYL and IMS schemas. Non-compliant one-to-many relationship types, for instance cyclic relationship types or relationship types between two master entity types, are most often expressed as relationship entity types (Figure 7-6). They can be recovered provided R1 can be proved to be one-to-one. Some one-to-many relationship types may be expressed implicitely as foreign keys, just like in CODASYL and IMS schemas. As before, recovering many-to-many or higher-degree relationship types is considered as of conceptual normalization concern.

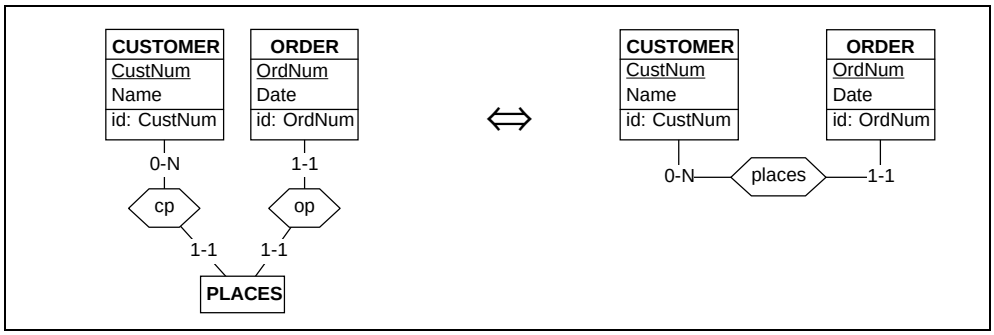


Figure 7-6: Recovering a one-to-many relationship type from a relationship entity type.

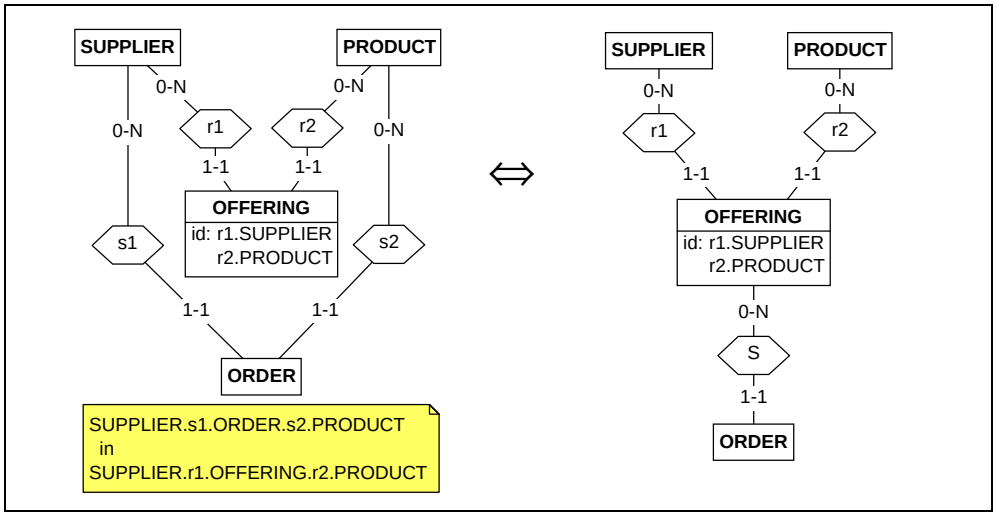


Figure 7-7: Recovering a 3-level hierarchy from a 2-level hierarchy. The constraint on the left-side schema states that the <SUPPLIER, PRODUCT> couple of each ORDER entity must correspond to an OFFERING entity.

The 2-level hierarchy constraint implies eliminating (1) non-hierarchical constructs, such as circuits, and (2) deep hierarchies. The technique of Figure 7-6 (reverse) is often used to move A one level up w.r.t. B (R is a common child of A and B, which are therefore at the same level). Conversely, interpreting child entity type R as a one-to-many relationship type between A and B will automatically recover the origin B-A hierarchy. However, other techniques can be used, such as that which is described in Figure 7-7, based on identifier substitution. It requires detecting the inclusion constraint that states that any (A,B) instance obtained from a D entity must identify a C entity (the notation S1oS2 expresses the relational composition of S1 and S2). One-to-many relationship type elimination through foreign key

(Figure 6-5) and entity type merging techniques (Figure 6-3) are also often observed.

7.5 Relational database reverse enginering

The relational model belongs to the basics of application development concepts, and need not be described in detail. From the reverse engineering point of view, we can identify two families of RDBMS, namely the first generation (all brands, such as Oracle up to version 6) and the second generation, known as SQL-2 compliant (almost all big names, including pioneer Digital RDB). A good illustration of the first one is Oracle version 5, which still is in use in many applications. It did not include primary or foreign keys, nor features such as database triggers, predicates or stored procedures. The only ways to encode integrity was through *Views with check option* and external techniques, including SQL-Forms pseudo-triggers (Section 3.4). The second generation proposes primary and foreign keys as standard, powerful check/assertion predicates, database triggers and stored procedures. Unfortunately, (1) modern SQL-2 engines still are servers for first generation applications and (2) new applications frequently are developed in such a way that they can use even first generation engines.

Relational Data structure extraction

There are two major sources for explicit constructs, namely SQL-DDL source texts and system tables. The later are more reliable since they describes the current state of the data structures, while the DDL texts can be obsolete or available as a series of incremental definitions instead of as a single text. Table 2 gives the main explicit structures extraction rules for SQL-2 DDL scripts.

Implicit structures derives from the restrictions of the model and from specific design practices: finding aggregated columns, implicit identifiers and implicit foreign keys form the main goals of this phase. However, all the other constraints must be addressed as well, as suggested in the following schema refinement script.

1. **Refine the column structure**
2. Find the implicit optional, compound or multivalued columns
3. Find the multiple column structures (rename and redefine)
4. **Find the missing table identifiers**
5. Interpret arrays as sets or lists
6. **Find the implicit foreign keys**
7. Find existence constraints among optional columns
8. Refine min-max cardinalities of multivalued fields
9. Identify non key functional dependencies and other redundancies
10. Find enumerated value domains and constraints on value domains

Relational Basic conceptualization

As far as data structures are concerned, the relational model is particularly poor: single-valued and atomic columns, no relationship types. Therefore, the main problems are to detect representations of multivalued attributes, compound attributes and relationship types.

Most one-to-many and one-to-one relationship types are represented through plain foriegn keys, and are easily recovered (Figure 6-5).

A multivalued attribute can be represented by a distinct table including a foreign key referencing the main table. This pattern is processed by first resolving the foreign key, then by integrating the attribute entity type as a mere attribute (Figure 7-6). Two other representation techniques are also frequently used, though less elegant, namely instantiation (**MultAtt-Serial**) and concatenation (**MultAtt-Single**).

The trace of an instantiation transformation can be detected by a structure of serial attributes, i.e., a sequence of attributes with the same type and length, and whose names present syntactical (EXP1, EXP2, etc) or semantical (JANUARY, FEBRUARY, etc) similarities (Figure 6-1).

The concatenation representation of a multivalued attribute consists in replacing the set of values by their concatenation, expressed as a single-valued attribute. Its domain appears as possibly made of a repeated simple domain (4 x char(12) in Figure 7-8). In fact, this concatenation transforms a set of values into an array or into a list.

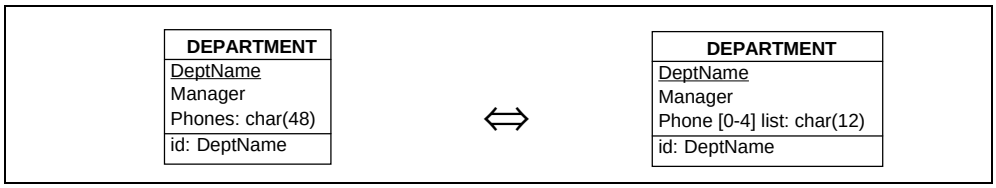


Figure 7-8: Representation of a concatenated attribute by a multivalued list attribute.

Array and list attributes may be considered undesirable, and pertaining to the implementation level. Indeed, these structures do not enforce uniqueness and add an ordering relation to the set of values. They are to be considered as poor representations of value sets. Therefore the semantics-preserving transformation of Figure 7-9 could be preferred, and should be applied when possible.

A compound attribute can be represented by concatenation, by attribute extraction as an entity type, or by disaggregation, to mention the most frequent techniques. Disaggregation can be detected by the presence of heterogeneous serial attributes whose names suggest a semantic correlation, for instance through a common prefix. Recovering this gouping is straightforward (Figure 6-2).

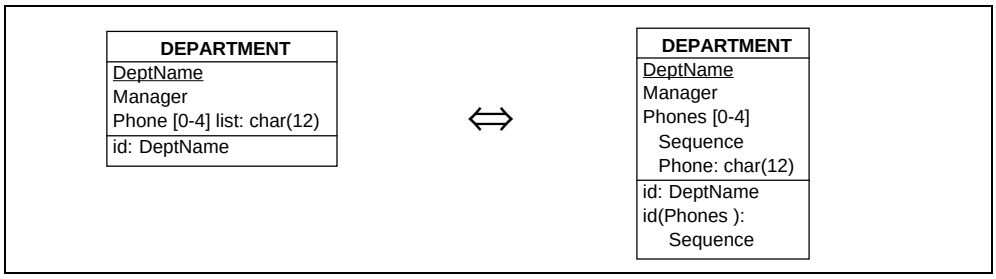


Figure 7-9: It can be wise to give a set interpretation of array and list structures.

7.6 Other standard DBMS reverse enginering

Database implemented in other popular DBMS such as ADABAS (Software A.G.) or DATACOM/DB (ADR, then Computer Assiociates) can be processed with the same techniques as those proposed in network, hierarchical and relational DBMS.

7.7 Object-oriented database reverse enginering

While database reverse engineering is getting mature, trying to recover the semantics of recent OO applications seems to trigger little interest so far. The reason is that the problem is overlooked because OO programs are supposed to be written in a clean and disciplined way, and based on state-of-the-art technologies which allow programmers to write code that is auto-documented, easy to understand and to maintain. The reality is more complex, and modern OO applications seem to suffer from the same problems as standard legacy systems as described in Chapter 1: weakness of the OODBMS models, implicit structures, optimized structures, awkward design and cross-model influence.

OO physical schemas generally are made of object classes, object attributes and methods (ignored her). An attribute (or instance variable) can be atomic or tuple (= compound), it can be single-valued or multivalued (set, bag, list). An important feature of OO-DBMS is that a class attribute can draw its values from another object class. Therefore, the value of such an attribute, which will be called *object-attribute*, can be an object or a collection of objects. An object-attribute expresses references to other objects, which in turn can include another object-attribute that references the former to form two sets of inverse references. The generic model presented in Chapter 2 is extended to the representation of this construct (Figure 7-10).

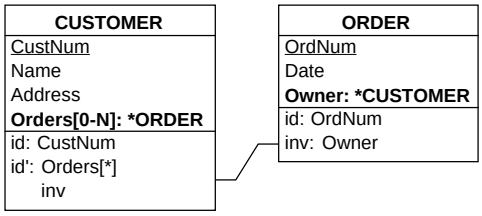


Figure 7-10: Object classes, object-attributes and inverse constraint. Note that Orders is a multi-valued identifier of CUSTOMER. Indeed, each order has one owner only, which means that an order identifies its customer.

The interpretation of these structures depends on the conceptual formalism used to express the conceptual schema. If this formalism does not include object-attributes (such is the case for OMT, UML and ODMG), they can be transformed into relationship types (Figure 7-11).

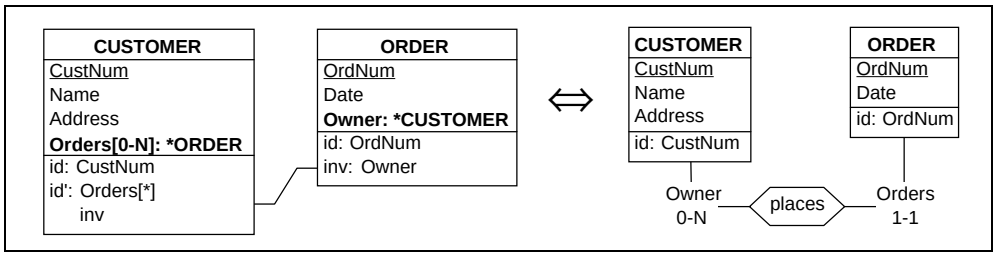


Figure 7-11: Transforming an object-attribute into a relationship type.

The main problems when reverse engineering OO databases come from the weaknesses of current OO-DBMS. Indeed, many of them still inherit from OO languages, and have a very poor set of integrity constraints. For instance, class identifiers, attribute identifiers, foreign keys and inverse relations cannot be explicitly declared, and must be found through all the elicitation techniques of the DS Extraction phase.

This topic will not be further developed in this section. See [Hainaut 1997b] and [Theodoros 1998] for instance for further detail.

Chapter 8

CASE technology for DBRE

Abstract

Database reverse engineering appears as a demanding activity according to several dimensions, ranging from the size and variety of the information sources to the complexity of the target data structures. In this section, we translate the characteristics of DBRE activities into CASE tool requirements and we present a representative CASE tool that includes specific DBRE-oriented features. The name CARE (for Computer-Aided Reverse Engineering) has once been suggested to identify the specific aspects of CASE. This term will be used in this section.

8.1 Requirements

We will try to capture the essence of DBRE in order to identify the needed specific features of a CASE tool that is to support reverse engineering activities. A more detailed analysis can be found in [Hainaut 1996b].

- The tool must allow very flexible working patterns, included unstructured and exploratory ones. A toolbox architecture is recommended. In addition, the tool must be be highly interactive.
- Specific functions should be easy to develop, even for one-shot use.
- The tool must include browsing and querying interfaces with a large variety of information sources. Customizable functions for automatic and assisted specification extraction should be available for each of them.
- It must provide sophisticated text analysis processors, including program understanding tools. The latter can be language independent, easy to customize and to program, and tightly coupled with the specification processing functions.
- The tool must include sophisticated name analysis and processing functions.
- A CARE tool must be a CASE tool as well. It includes a large set of functions, including those required for forward engineering.
- It easily communicates with the other development tools of the organization, such as other CASE tools, Data dictionaries or DMS, e.g., via integration hooks or communication with a common repository.
- The specification model must be wide-spectrum, and allow the representation and the processing of inconsistent components of different abstraction levels.
- The specification model and the basic techniques offered by the tool must be DMS-independent, and therefore highly generic.
- The CARE tool must provide several ways of viewing both source texts and abstract structures (schemas). Multiple textual and graphical views, summary and fine-grained presentations must be available.
- The CARE tool must provide a rich set of schema transformation techniques. In particular, this set must include operators which can undo the transformations commonly used in empirical database designs.
- The repository of the CARE tool must record all the links between the schemas at the different levels of abstraction. More generally, the tool must ensure the *traceability* of the reverse engineering processes.

Now, we discuss some technical aspects of CARE tools and illustrate them through the DB-MAIN environment that addresses some of these requirements. This CASE environment, developed since 1991, is intended to support most database applications engineering process, including reverse engineering[1].

8.2 Project and document representation and management

Reverse engineering projects can use a large number of documents (coping with hundreds of schemas and thousands of program source files is not unfrequent) that are involved in as many engineering processes. These documents and activities must be represented and documented. An important aspect of project management is activity tracing, through which the history of the project is recorded, and can be searched, queried and processed, at various levels of detail (Figure 9-13). This history should ensure a complete forward and backward traceability of the project.

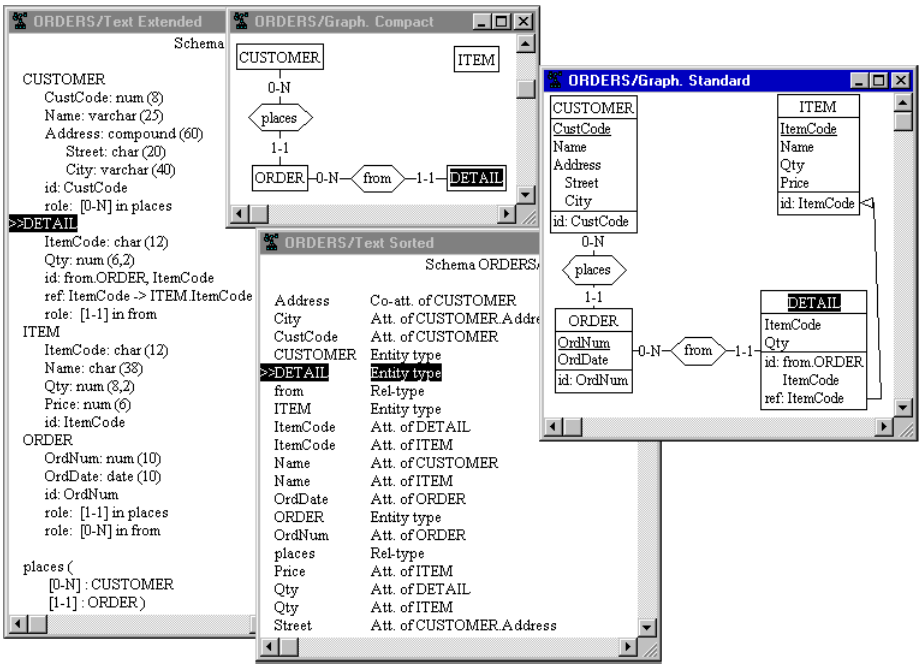


Figure 8-1: Four representations of the same schema: text extended (left), text sorted (bottom), graphical compact (top) and graphical standard (right).

The specification documents, both formal and textual, at their different evolution states must be available for examination, analysis and modification. The tool must be able to process large schemas (e.g., 500 record types with 10,000 fields) and texts (e.g., beyond 100,000 loc per program unit), and to let the analyst examine and manipulate them through different way

1. An education version of the tool, together with related educational, technical and scientific material, can be obtained at `http://www.info.fundp.ac.be/libd`.

of viewing. For instance, a graphical representation of a schema allows an easy detection of certain structural patterns, but is useless to analyse name correspondances and similarities, an activity that requires textual presentations (Figure 8-1). The same can be said of program representation.

8.3 Support for the data structure extraction process

According to Chapter 5, a CARE tool must offer a collection of analyzers for the main information sources. We mention some examples here below.

First of all, the tool must include *DDL extractors* for popular DMS, such as COBOL, RPG, IMS, CODASYL DDL and SQL. For missing DMS, it should be easy to write specific extractors. These processors create an abstract schema expressing the physical concepts of a DDL text or of a data dictionary.

Program texts will be searched for instances of specific *patterns* ranging from the simplest ones (locate all the "WRITE" statements) to complex syntactic structures such as that of Figure 5-4.

Dataflow and dependency analyzers build the relational graph of a program according to a set of user defined relations. For instance, the COBOL assignment relation defined by pattern "MOVE @var1 TO @var2" yields the equality graph of the program. Then, this graph will be queried for specific objects. The result can be presented graphically or in context, by colouring the result statements in the program. These graphs must be coupled with the database schema the program is concerned with.

Reducing a large program to the set of statements (i.e., the *slice*) that concern specific variables at a given program point requires a *program slicer*. This processor must be parametrizable in order to control the precision of the result, and must be coupled to the database schema in concern.

The physical schema itself can give structural information that contribute to the identification of hidden constraints and structures. Hence the importance of a *schema analyzer* which can be controlled by structural predicates.

Foreign keys are the most important structures to elicit in practically all schemas. A dedicated foreign key analyzer must help apply the most common heuristics (Figure 8-2).

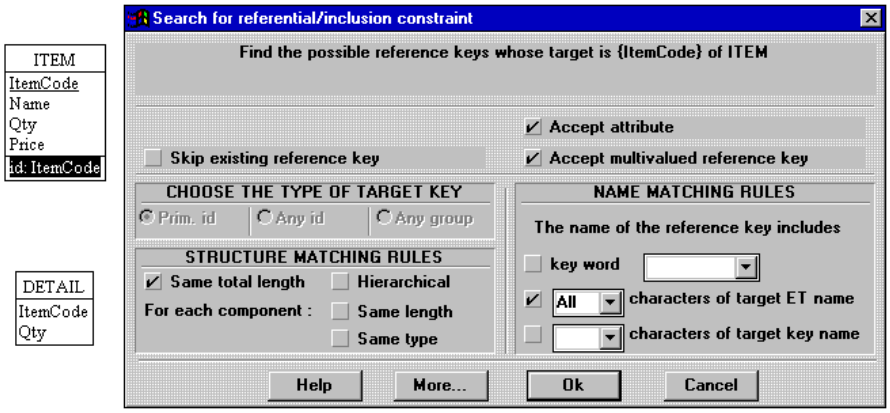


Figure 8-2: Control panel of the search engine of the Reference key assistant of DB-MAIN. Considering a selected identifier (left), it has been asked to find all the attributes, possibly multivalued, that have the same total length, and whose name includes all the characters of the target entity type, i.e. "ITEM". The master panel will then show all the candidate foreign keys, including *ItemCode*, and propose to define one or several of them.

Name analysis can bring important hints about hidden structures and synonyms, among others. So, special processors will search schemas for name patterns and display names in sorted order for visual examination of name sequences. *Name processing* consists in changing names to make them more expressive and /or standardized (e.g., expanding "INV_" into "Invoice-", or capitalize uppercase names).

Schema integration requires intelligent processors that are able to detect semantic correspondences and to merge schemas while avoiding redundancies. Physical integration processors can exploit layout similarities and dissimilarities among a set of record/field structures in order to propose a common view of them.

Data analysis can be an important way to get hints and to evaluate hypotheses. A CARE tool can read foreign data from a database, or can generate queries and small application programs that report about candidate data properties such as uniqueness, inclusion (e. g., in foreign keys), nullable fields, etc.

8.4 Support for the data structure conceptualization process

This process, described in Chapter 6, heavily relies on *transformation techniques*. For some fine-grained reasonings, precise *chirurgical* transformations have to be carried out on individual constructs. This is a typical way of working in *de-optimization* activities. In other cas-

es, all the constructs that meet a definite precondition have to be transformed (e.g., all standard foreign keys are replaced with binary relationship types). Finally, some heuristics can be identified and materialized into a transformation plan. More precisely, the following three levels of transformation must be available.

1. *Elementary transformations*. Transformation T is applied to object O. With these tools, the user keeps full control on the schema transformation since similar situations can be solved by different transformations; e.g., a multivalued attribute can be transformed in a dozen of ways. The CARE tool must offer a rich toolbox of elementary transformations.
2. *Global transformations*. Transformation T is applied to all the objects of a schema that satisfy predicate P. Such transformations can be carried out through a processor that allows the analyst to define T and P independently. *Examples*: replace all *single-component one-to-one relationship types* with *foreign keys*; replace all *multivalued compound attributes* with *entity types*.

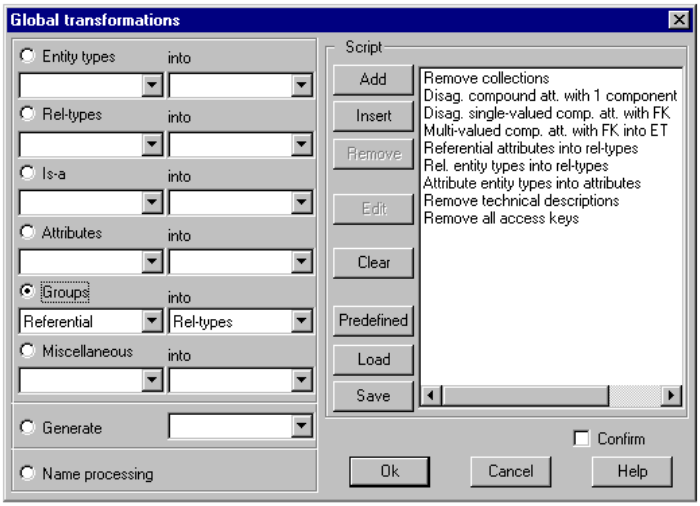


Figure 8-3: The *Basic Global transformation* assistant of DB-MAIN showing a simple transformation script for COBOL conceptualization. The *Advanced Global transformation* assistant includes user-defined predicates, filters and loop control structures as well as user-defined predicates and operations libraries.

3. *Model-driven transformations*. All the constructs of a schema that do not comply with a given model are processed through a transformation plan. The CARE tool must allow analysts to develop their own transformation plans (Figure 8-3).

Name processing and the schema integration will be used for refining the naming patterns and merging conceptuel schemas if needed. Report generators will make the final conceptual specifications available to their users while bridges with foreign CASE tools and Data Dictionaries will distribute selected specifications among the information resources managers of

the organization.

8.5 The DB-MAIN CASE environment

DB-MAIN is a general purpose CASE and meta-CASE environment which includes database reverse engineering and program understanding tools. Its main goal is to support all the database application engineering processes, ranging from database development to system evolution, migration and integration. In this scope, mastering DBRE is an essential requirement. The environment has been developed by the *Database Engineering Application Laboratory* (LIBD) of the University of Namur, as part of the DB-MAIN project. Extensions have been (and are being) developed towards federated database methodology through the InterDB project [Thiran 1998], methodological support for temporal databases (TimeStamp project [Detienne 2001]) and XML engineering (Data migration projet [Delcroix 2001]). More specifically, it includes the following functions, components and capabilities:

- specifications management: access, browsing, creation, update, copy, analysis, memorizing;
- representation of the project history: processes, schemas, views, source texts, reports, generated programs and their relationships;
- a generic, wide-spectrum, representation model for conceptual, logical and physical objects; accepts both entity-based and object-oriented specifications; schema objects and text lines can be selected, marked, aligned and colored;
- semantic and technical annotations can be attached to each specification object;
- multiple views of the specifications (4 hypertexts and 2 graphical views); some views are particularly intended for very large schemas; both entity-based and object-oriented schemas can be represented;
- a toolbox of about thirty semantics-preserving transformational operators which provide a systematic way to carry out such activities as conceptual normalization, or the development of optimized logical and physical schemas from conceptual schemas, and conversely (i.e., reverse engineering);
- code generators; report generators;
- code parsers extracting physical schemas from SQL, COBOL, CODASYL, RPG and IMS source programs;
- interactive and programmable text analysers which can be used, a.o., to detect complex programming *clichés* in source texts, to build dataflow and dependency diagrams, and to compute program slices [Henrard 1998];
- a name processor to search a schema for name patterns and to clean, normalize, convert or translate the names of selected objects;

- a history manager which records the engineering activities of the analyst, and which makes their further replay possible;
- import and export of specifications;
- a series of assistants, which are expert modules in specific kinds of tasks, or in classes of problems, and which are intended to help the analyst in frequent, tedious or complex activities. It allows the analyst to develop scripts which automate frequent processes. A library of predefined scripts is provided for the most frequent activities. Six assistants are available at present: Basic global transformation (Figure 8-3), Advanced global transformation, Schema analysis, Schema integration, Text analysis and Reference key analysis (Figure 8-2).

No tools can be claimed to solve all current and future problems. Therefore, DB-MAIN also includes a meta-development environment that allows administrators and method engineers to extend, specialize, and combine the existing concepts and functions, and to develop new ones. Extension can be performed in four ways.

First, the generic model itself can be enriched through several techniques such as meta-properties, semi-formal properties, stereotypes, and triggers associated with repository primitives.

Secondly, as already mentioned, several assistants include a script facility through which fragments of method can be defined and stored, such as transformation, name processing, schema analysis or text pattern definition.

Thirdly, specific methods can be defined, and enforced by the tool. A method is defined by an MDL (Method Definition Language) script, compiled as a part of the repository, then enacted by the method engine.

Finally, new processors, such as specific report and code generators, DDL analyzers, or specifications checkers, can be developed in *Voyager 2*. This language allows CASE engineer (analyst or method engineer) to develop new functions which will be seamlessly incorporated in the tool without resorting to C++ programming. It is a complete 4th-generation language which offers predicative access to the repository, easy analysis and generation of external texts, definition of recursive functions and procedures, and a sophisticated list manager. It makes the rapid development of complex functions possible.

8.6 State of the art in database CARE tools

In the early nineties, an increasing number of DBRE CASE tools were proposed to practitioners [Rock 1990]. Almost every standard CASE tool was enriched with so-called reverse engineering processors. The state of the art has completely changed at the present time. Indeed, experience with large DBRE project has shown that the mythical *Automatic Database Reverse Engineering*, sometimes based on some intelligent algorithms and heuristics, has now completely faded away. For instance, one of biggest names in CASE tools has progessively reduced the scope of its once largely advertized reeingineering environment into a mere help

in schema recovery. Most current CASE tools include a DDL extractor, a foreign key discoverer (based on simplistic naming assumptions) and a primitive *foreign key/relationship type* transformer. Such tools can only bring some help in highly disciplined (and therefore very unfrequent) database designs.

On the contrary, DBRE appears as an interactive decision-based intellectual, and often manual, activity. Current tools can help in extracting first-cut schemas and in maintaining schema documentation in a graphical form. With them, almost all complex analysis and transformation processes must be carried out manually. This is the way most DBRE projects are performed today.

Chapter 9

A case study: Database Migration

Abstract

This section describes a small case study which is presented as a part of a migration project in which a set of COBOL files are to be converted into relational tables. The objective of the exercise is to produce a relational schema which translates as faithfully as possible the semantics of those source files. As expected for such a small case study, not all the problems, all the techniques and all the reasonings will be illustrated. We will mainly focus on program and schema analysis.

20/5/2002

9.1 Introduction

The source data structures appear in a small COBOL program which uses three files. These files are to be converted into a relational databases in such a way that all the semantics of the source data structures are translated into relational structures. This can be done in two steps: first we elaborate a conceptual schema of the three files, then we translate this schema into relational structures (Figure 9-1).

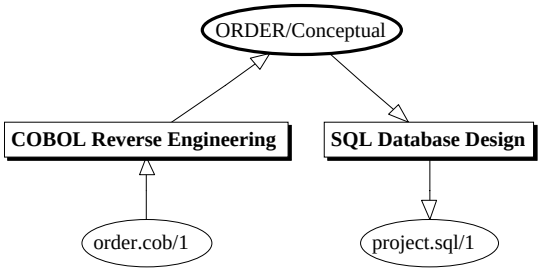


Figure 9-1: First level history of the whole project.

9.2 Project preparation

The only source of information that will be considered is the COBOL program listed in Chapter 5 (named order.cob). The main processes and documents of the project, which will be carried out with the help of a CASE tool, are described in Figure 9-1 and Figure 9-14.

9.3 Data structure extraction

The objective is to recover the complete logical schema comprising the explicit constructs expressed in the data structures declaration statements of the program(s) as well as the implicit constructs burried, essentially, in the procedural statements.

DMS-DDL code extraction

This operation is carried out by a COBOL parser which extracts the file and record type descriptions, and expresses them as a *raw physical* schema in the repository of the CASE tool

(Figure 9-2).

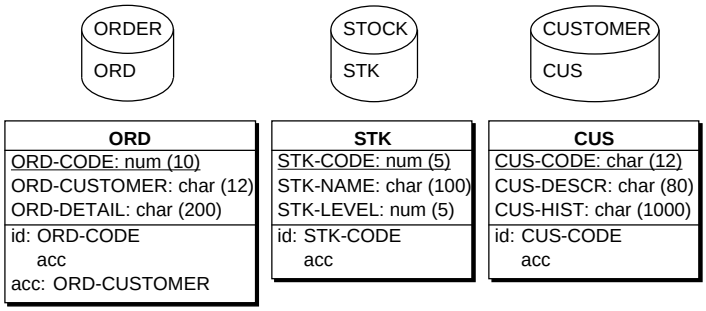


Figure 9-2: The raw physical schema: explicit file, record, field and key description.

Each record type is represented by a physical entity type, and each field by a physical attribute. Record keys are represented by identifiers when they specify uniqueness constraints and by access keys when they specify indexes. Files are represented as physical entity collections.

Schema refinement

This schema will be refined through an in-depth inspection of the way in which the program uses and manages the data. Through this process, we will detect additional structures and constraints which were not explicitly declared in the file/record declaration sections, but which were expressed in the procedural code and in local variables. We will consider four important constructs, namely Field structure, Foreign keys, Multivalued field identifiers, and Field cardinality.

- **Field structure**

Observation: some fields are unusually long (CUS-DESCR, CUS-HIST, ORD-DETAIL, STK-NAME). Could they be further refined? Let us consider CUS-DESCR first. We build the variable dependency graph, which summarizes the dataflow concerning CUS-DESCR (statements <1> and <1'>):

CUS.CUS-DESCR ◄——► **DESCRIPTION**

This graph clearly suggests that CUS-DESCR and DESCRIPTION share the same values, and should have the same structure as well, i.e.:

```
01 DESCRIPTION.
   02 NAME PIC X(20).
   02 ADDRESS PIC X(40).
   02 FUNCTION PIC X(10).
   02 REC-DATE PIC X(10).
```

This structure is associated with the field CUS-DESCR in the physical schema. We proceed in the same way for CUS-HIST, ORD-DETAIL and STK-NAME (Figure 9-3). The analysis shows that the last one need not be refined.

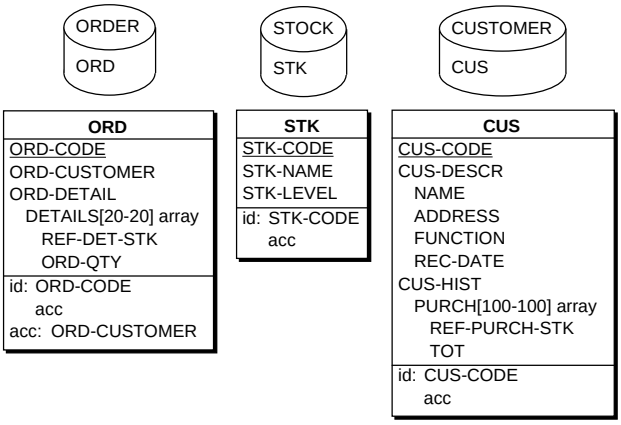


Figure 9-3: Physical schema (refined): the concatenated compound fields ORD-DETAIL, CUS-DESCR and CUS-HISTORY have been decomposed.

- **Foreign key elicitation**

There should exist reference links among these record types. Let us examine the field ORD-CUSTOMER for instance. We observe that:

- its name includes the name of a file (CUSTOMER);
- it has the same type and length as the record key of CUSTOMER;
- it is supported by an access key (i.e., an index);
- its dependency graph shows that it receives its values from the record key of CUSTOMER <4>:

CUS.CUS-CODE ⟶ **ORD.ORD-CUSTOMER**

- its usage pattern shows (through a program slice) that, before moving it to the ORD record to be stored, the program checks that ORD-CUSTOMER value identifies a stored CUS record (Figure 9-4).

These are five positive evidences contributing to making us confident that ORD-CUSTOMER is a foreign key. We decide to confirm the hypothesis. In the same way, we conclude that:

- ORD-DETAIL.DETAILS.REF-DET-STK is a multi-valued foreign key to STOCK. Here the REF part of the name suggests the referential function of the field.
- CUS-HIST.PURCH.REF-PURCH-STK is a multivalued foreign key to STOCK.

Now the schema looks like that in Figure 9-5.

```
      NEW-ORD.
         ...
         MOVE 1 TO END-FILE.
         PERFORM READ-CUS-CODE UNTIL END-FILE = 0.
         ...
         MOVE CUS-CODE TO ORD-CUSTOMER.
         ...
         WRITE ORD INVALID KEY DISPLAY "ERROR".
         ...
      READ-CUS-CODE.
         ACCEPT CUS-CODE.
         MOVE 0 TO END-FILE.
         READ CUSTOMER INVALID KEY
           DISPLAY "NO SUCH CUSTOMER"
           MOVE 1 TO END-FILE
         END-READ.
```

Figure 9-4: The program slice Π*(order.cob,ORD.ORD-CUSTOMER,line("WRITE ORD"))* that shows what happens before writing an ORD record in the ORDER file.

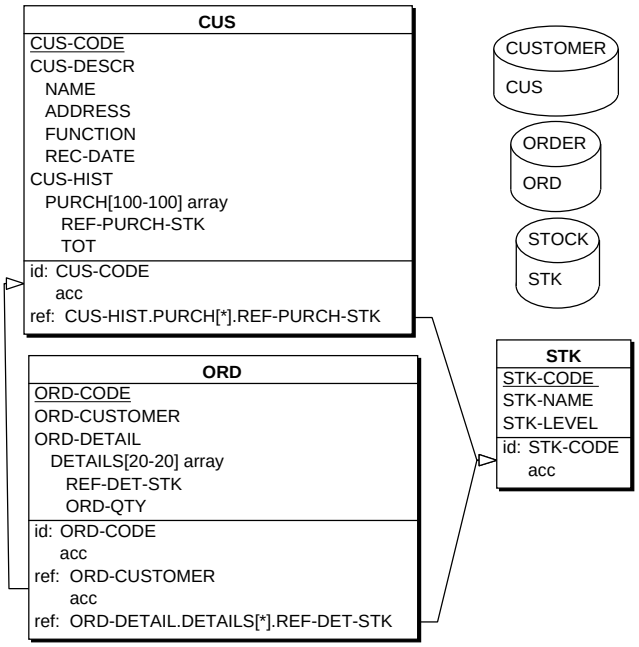


Figure 9-5: Physical schema (refined): three foreign keys have been made explicit.

- **Elicitation of identifiers of multivalued fields**

Compound multivalued fields in COBOL records often have an *implicit identifier*. This means that multivalued field **F** has a (set of) subfield(s) **I** that makes the values of **F** unique on **I**. The schema includes two candidate multivalued fields: ORD-DETAIL.DETAILS and CUS-HIST.PURCH. By examining the way in which these fields are searched and managed, we isolate the program fragment of Figure 9-6, that has been extracted from the program slice Π(order.cob,ORD.ORD-DETAIL.DETAILS,line("WRITE ORD")).

```
MAIN.                                   READ-PROD-CODE.
  PERFORM PROCESS                         PERFORM UPDATE-ORD-DETAIL.
  UNTIL CHOICE = 0.                     UPDATE-ORD-DETAIL.
                                          MOVE 1 TO NEXT-DET.
PROCESS.                                  PERFORM UNTIL
  PERFORM NEW-ORD.                          REF-DET-STK(NEXT-DET)
                                              = PROD-CODE
NEW-ORD.                                  OR IND-DET = NEXT-DET
  SET IND-DET TO 1.                       ADD 1 TO NEXT-DET
  PERFORM READ-DETAIL                   END-PERFORM.
    UNTIL END-FILE = 0                  IF IND-DET = NEXT-DET
    OR IND-DET = 21.                      MOVE PROD-CODE
  WRITE ORD                               TO REF-DET-STK(IND-DET)
    INVALID KEY DISPLAY "ERROR".          SET IND-DET UP BY 1
                                        ELSE
READ-DETAIL.                              DISPLAY "ERROR : ...".
  PERFORM READ-PROD-CODE.
```

Figure 9-6: Program fragment that suggests that field REF-DET-STK is an identifier for ORD-DETAIL.DETAILS. It also gives information on the exact cardinalities of ORD-DETAIL.DETAILS. This fragment has been extracted from the slice .

It derives from this code section that the LIST-DETAIL.DETAILS array of a record will never include twice the same REF-DET-STK value. Therefore, this field is the local identifier of this array, and of ORD-DETAIL.DETAILS as well. Through the same reasoning, we are suggested that REF-PURCH-STK is the identifier of LIST-PURCHASE.PURCH array. These findings are shown in Figure 9-7.

- **Refinement of the cardinality of multivalued attributes**

The multivalued fields have been given cardinality constraints derived from the *occurs* clauses. The latter give the maximum cardinality, but tell nothing about the minimum cardinality.

Storing a new CUS record generally implies initializing each field, including CUS-HIST.PURCH. This is done through the INIT-HIST paragraph (line <13>), in which the REF-DET-STK is set to 0. Furthermore, the scanning of this list stops when 0 is encountered (line <7>). The conclusion is clear: there are from 0 to 100 elements in the list. A similar analysis of the fragment of Figure 9-6 leads to refine the cardinality of ORD-DETAIL.DETAILS. Hence the schema of Figure 9-7.

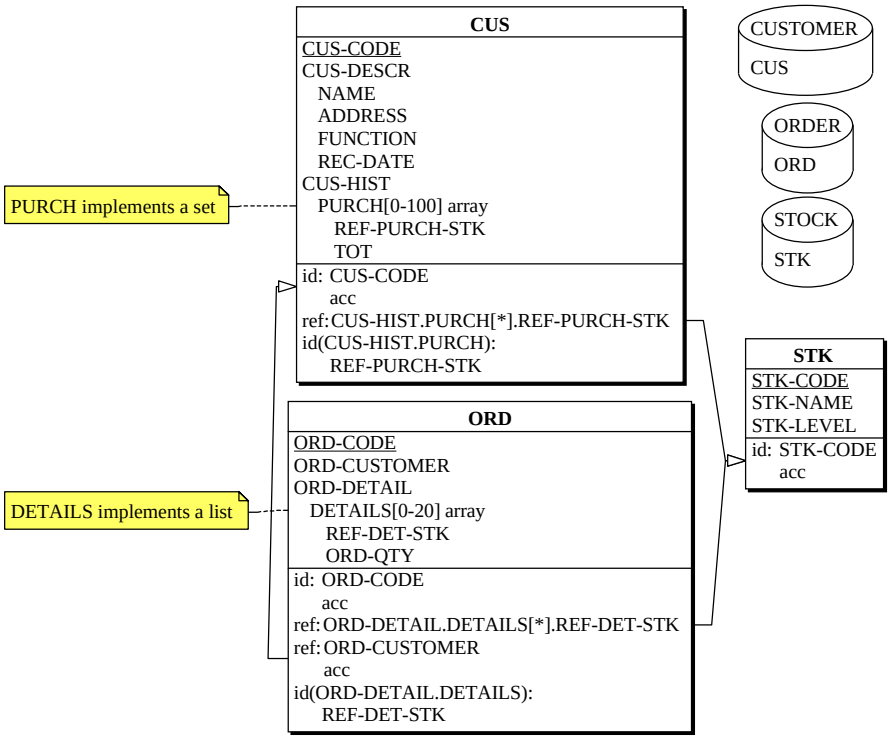


Figure 9-7: Physical and logical schema (completed): the identifiers of multivalued attributes have been detected and the minimum cardinalities have been set to their intended value.

Looking for more significant names

Program and schema analysis has given us an opportunity to find more informative names for some schema objects.

1. The file names appear to be more explicit than the name of their respective record types: CUSTOMER should be used for CUS, STOCK for STK and ORDER for ORD.
2. A program variable can have a better name than the field it receives its value from. So, CUS-DESCR could be renamed DESCRIPTION.

Interpreting arrays

Since the array is the most straighforward implementation of multivalued attributes, we should identify those which actually implement lists, bags and sets. Data analysis and usage pattern analysis can be used. We observe that:

1. *for CUS*: the ordering of values of PURCH and the possibility to leave cells empty are

irrelevant, so that this array implements a mere set;

2. *for ORD*: the order of DETAILS values could be significant but there is no means to leave a cell empty; this array seems to implement a list (in fact, a unique list).

9.4 Data structure conceptualization

Schema preparation

The schema obtained so far describes the complete COBOL data structures, including both implicit and explicit constructs. Before trying to recover the conceptual schema, we clean the current schema (Figure 9-8).

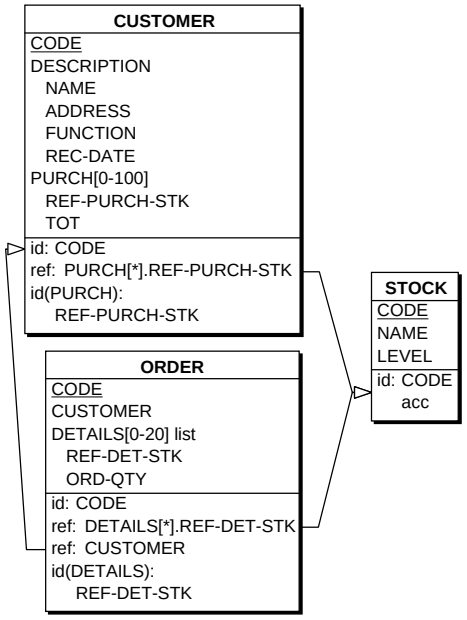


Figure 9-8: The (improved) logical schema.

- *Discarding technical constructs*. The physical constructs, namely the files and the access keys, are no longer useful, and are removed.
- *Name processing (1)*. The fields of each record type are prefixed with a common short-name identifying their record type. This is a common programming trick that provides shorter names but adds no semantics. We trim them out.
- *Name processing (2)*. The new names suggested by schema and program analysis are assigned to the schema objects.

- *Uselessly compound fields*. Compound fields CUS.CUS-HIST and ORD.ORD-DETAIL have one component only, and can be disaggregated without structural or semantic loss.
- *Interpreting arrays*. As observed in the previous phase, CUSTOMER.PURCH is a set while ORDER.DETAILS is a list. We modify the schema accordingly.

Basic conceptualization

- **Maximum cardinalities**

Are the maximum cardinalities 100 and 20 of real semantic value, or do they simply express obsolete technical limits from the legacy system? We take for true that there cannot be more than twenty lines of detail in an order but that a customer can have any number of purchases associated with him/her. The cardinality of DETAILS is left as is, but the cardinality of PURCH is changed to [0-N].

- **Expressing non-set multivalued attributes**

Bags and lists (as well as arrays, if any) are expressed by set structures.

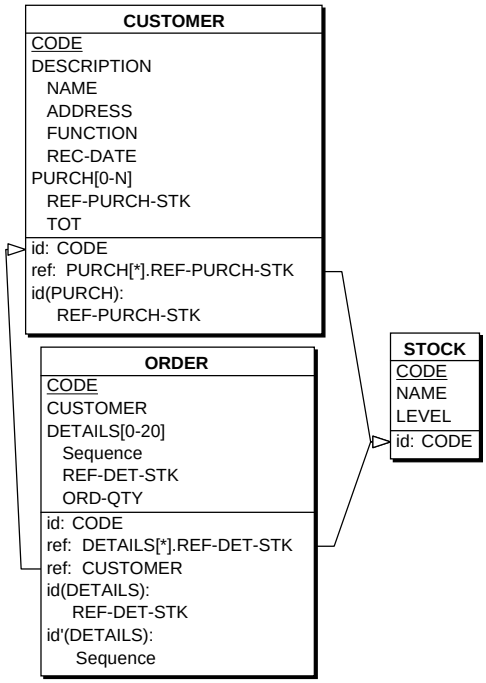


Figure 9-9: The maximum cardinalities have been evaluated for generalization and non-set multi-valued attributes have been transformed.

- **Complex attributes**

The attributes CUSTOMER.PURCH and ORDER.DETAILS have a particularly complex structure: they are compound, they are multivalued, they have local identifiers and they include a foreign key. They obviously suggest a typical COBOL trick to represent dependent entity types. This very efficient technique consists in representing such entity types by embedded multivalued fields. We transform the latter into entity types (Figure 9-10).

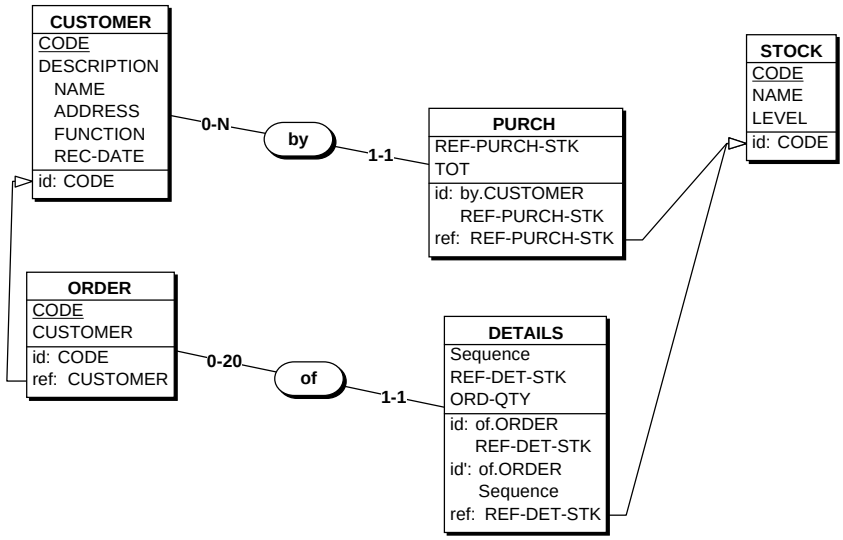


Figure 9-10: Making dependent entity types explicit.

- **Foreign key interpretation**

  The foreign keys are the most obvious traces of the COBOL translation of one-to-many relationship types (Figure 9-11).

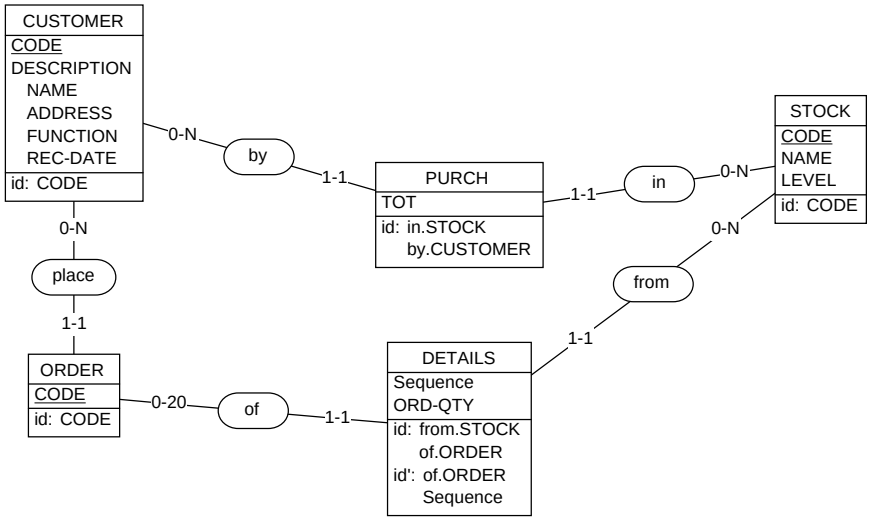


Figure 9-11: Untranslating foreign keys into relationship types.

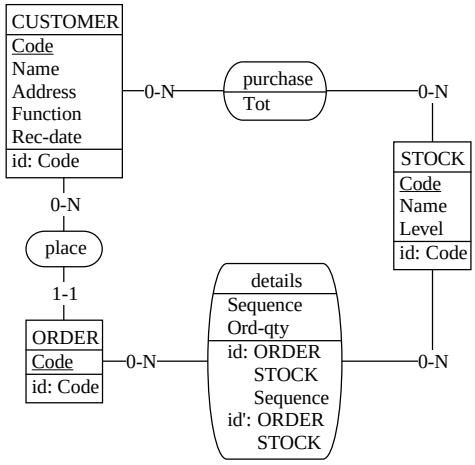


Figure 9-12: A normalized conceptual schema.

Conceptual normalization

We will only mention three elementary problems to illustrate the process (Figure 9-12).

- *Relationship type entity types*. PURCH and DETAILS could be perceived as mere relationships, and are transformed accordingly.

- *Names*. Now the semantics of the data structures have been elicited, and better names can be given to some of them. For instance, PURCH is given the full-name **purchase**.
- *Unneeded aggregation.* Attribute DESCRIPTION in CUSTOMER is an artificial aggregate which can be dismantled without loss of semantics.

9.5 Database conversion

To complete the exercice, let us develop a new relational database schema from the conceptual specifications. The process is fairly standard, and includes the Logical design and the Physical design phases. Due to the size of the problem, they are treated in a rather symbolic way.

Logical design

Transforming this schema into relational structures is fairly easy: we express the complex relationship types **detail** and **purchase** into entity types, then we translate the one-to-many relationship types into foreign keys. The resulting schema comprises flat entity types, identifiers and foreign keys. It can be considered a logical relational schema (Figure 9-13).

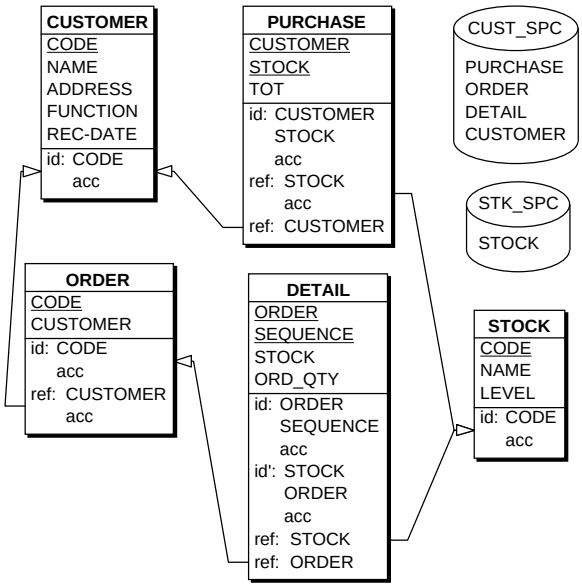


Figure 9-13: The physical relational schema.

Physical design and coding

We reduce this phase to processing the names according to SQL standard (e.g., all the names in uppercase, no "-", no reserved words, etc.) and defining the record collections (dbspaces) and the access keys (indexes) which support identifiers and foreign keys (Figure 9-13). As a symbolic touch of optimization, we remove all the indexes which are a prefix of another index (i.e., no index on PURCHASE.CUSTOMER and on DETAILS.STOCK). Generating the SQL script that encodes this schema is straighforward.

Project history

To summarize the process, we show the history of the project in Figure 9-14.

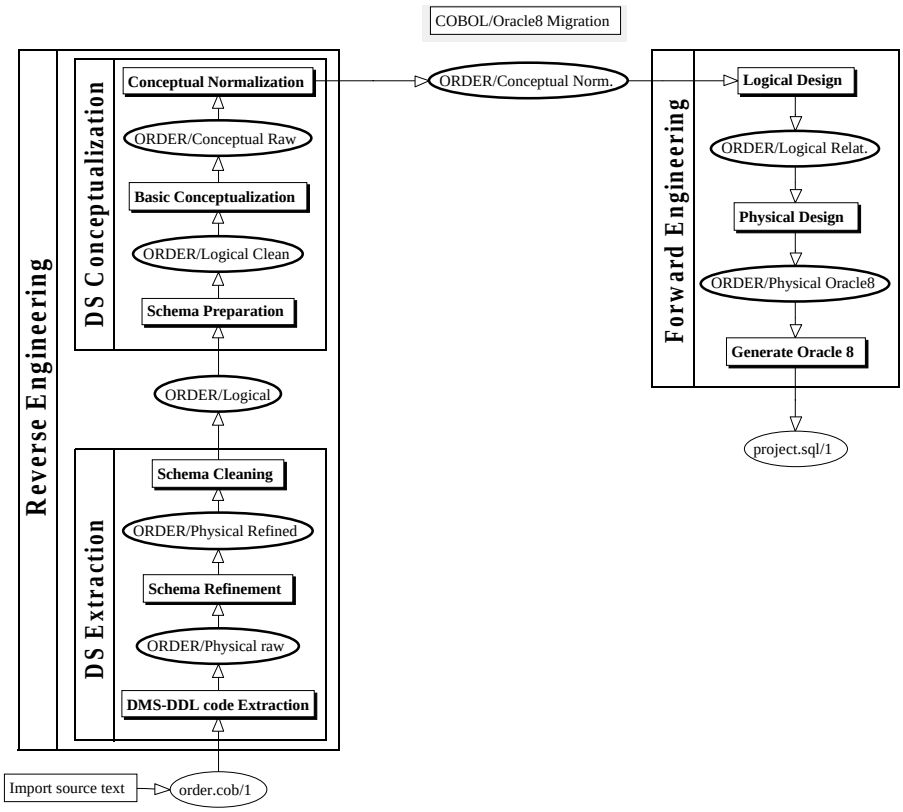


Figure 9-14: Project history depicting the main engineering processes.

9.6 The COBOL source code

```
IDENTIFICATION DIVISION.
PROGRAM-ID. C-ORD.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
   SELECT CUSTOMER
      ASSIGN TO "CUSTOMER.DAT"
      ORGANIZATION IS INDEXED
      ACCESS MODE IS DYNAMIC
      RECORD KEY IS CUS-CODE.
   SELECT ORDER
      ASSIGN TO "ORDER.DAT"
      ORGANIZATION IS INDEXED
      ACCESS MODE IS DYNAMIC
      RECORD KEY IS ORD-CODE
      ALTERNATE RECORD KEY
         IS ORD-CUSTOMER
         WITH DUPLICATES.
   SELECT STOCK
      ASSIGN TO "STOCK.DAT"
      ORGANIZATION IS INDEXED
      ACCESS MODE IS DYNAMIC
      RECORD KEY IS STK-CODE.

DATA DIVISION.
FILE SECTION.
FD CUSTOMER.
01 CUS.
   02 CUS-CODE PIC X(12).
   02 CUS-DESCR PIC X(80).
   02 CUS-HIST PIC X(1000).
FD ORDER.
01 ORD.
   02 ORD-CODE PIC 9(10).
   02 ORD-CUSTOMER PIC X(12).
   02 ORD-DETAIL PIC X(200).
FD STOCK.
01 STK.
   02 STK-CODE PIC 9(5).
   02 STK-NAME PIC X(100).
   02 STK-LEVEL PIC 9(5).

WORKING-STORAGE SECTION.
01 DESCRIPTION.
   02 NAME PIC X(20).
   02 ADDRESS PIC X(40).
   02 FUNCTION PIC X(10).
   02 REC-DATE PIC X(10).
01 LIST-PURCHASE.
   02 PURCH OCCURS 100 TIMES
      INDEXED BY IND.
      03 REF-PURCH-STK PIC 9(5).
      03 TOT PIC 9(5).
01 LIST-DETAIL.
   02 DETAILS OCCURS 20 TIMES
      INDEXED BY IND-DET.
      03 REF-DET-STK PIC 9(5).
      03 ORD-QTY PIC 9(5).

01 CHOICE PIC X.
01 END-FILE PIC 9.
01 END-DETAIL PIC 9.
01 EXIST-PROD PIC 9.
01 PROD-CODE PIC 9(5).

01 TOT-COMP PIC 9(5) COMP.
01 QTY PIC 9(5) COMP.
01 NEXT-DET PIC 99.

PROCEDURE DIVISION.
MAIN.
   PERFORM INIT.
   PERFORM PROCESS UNTIL CHOICE = 0.
   PERFORM CLOSING.
   STOP RUN.
INIT.
   OPEN I-O CUSTOMER.
   OPEN I-O ORDER.
   OPEN I-O STOCK.
PROCESS.
   DISPLAY "1 NEW CUSTOMER".
   DISPLAY "2 NEW STOCK".
   DISPLAY "3 NEW ORDER".
   DISPLAY "4 LIST OF CUSTOMERS".
   DISPLAY "5 LIST OF STOCKS".
   DISPLAY "6 LIST OF ORDERS".
   DISPLAY "0 END".
   ACCEPT CHOICE.
   IF CHOICE = 1
      PERFORM NEW-CUS.
   IF CHOICE = 2
      PERFORM NEW-STK.
   IF CHOICE = 3
      PERFORM NEW-ORD.
   IF CHOICE = 4
      PERFORM LIST-CUS.
   IF CHOICE = 5
      PERFORM LIST-STK.
   IF CHOICE = 6
      PERFORM LIST-ORD.
```

```
CLOSING.
   CLOSE CUSTOMER.
   CLOSE ORDER.
   CLOSE STOCK.

NEW-CUS.
   DISPLAY "NEW CUSTOMER:".
   DISPLAY "CUSTOMER CODE?"
       WITH NO ADVANCING.
   ACCEPT CUS-CODE.
   DISPLAY "NAME DU CUSTOMER: "
      WITH NO ADVANCING.
   ACCEPT NAME.
   DISPLAY "ADDRESS OF CUSTOMER: "
      WITH NO ADVANCING.
   ACCEPT ADDRESS.
   DISPLAY "FUNCTION OF CUSTOMER: "
      WITH NO ADVANCING.
   ACCEPT FUNCTION.
   DISPLAY "DATE: "
      WITH NO ADVANCING.
   ACCEPT REC-DATE.
   MOVE DESCRIPTION TO CUS-DESCR.   <1>
   PERFORM INIT-HIST.
   WRITE CLI
      INVALID KEY DISPLAY "ERROR".

LIST-CUS.
   DISPLAY "LISTE DES CUSTOMERS".
   CLOSE CUSTOMER.
   OPEN I-O CUSTOMER.
   MOVE 1 TO END-FILE.
   PERFORM READ-CUS
      UNTIL END-FILE = 0.

READ-CUS.
   READ CUSTOMER NEXT
   AT END MOVE 0 TO END-FILE
   NOT AT END
      DISPLAY CUS-CODE
      DISPLAY CUS-DESCR
      DISPLAY CUS-HISTORY.

NEW-STK.
   DISPLAY "NEW STOCK".
   DISPLAY "PRODUCT NUMBER: "
      WITH NO ADVANCING.
   ACCEPT STK-CODE.
   DISPLAY "NAME: " WITH NO ADVANCING.
   ACCEPT STK-NAME.
   DISPLAY "LEVEL: " WITH NO ADVANCING.
   ACCEPT STK-LEVEL.
   WRITE STK
       INVALID KEY DISPLAY "ERROR ".

LIST-STK.
   DISPLAY "LIST OF STOCKS ".
   CLOSE STOCK.
   OPEN I-O STOCK.
   MOVE 1 TO END-FILE.
   PERFORM READ-STK UNTIL END-FILE = 0.

READ-STK.
   READ STOCK NEXT
      AT END MOVE 0 TO END-FILE
      NOT AT END
         DISPLAY STK-CODE
         DISPLAY STK-NAME
         DISPLAY STK-LEVEL.

NEW-ORD.
   DISPLAY "NEW ORDER".
   DISPLAY "ORDER NUMBER: "
      WITH NO ADVANCING.
   ACCEPT ORD-CODE.
   MOVE 1 TO END-FILE.
   PERFORM READ-CUS-CODE
      UNTIL END-FILE = 0.
   MOVE CUS-DESCR TO DESCRIPTION.  <1'>
   DISPLAY NAME.
   MOVE CUS-CODE TO ORD-CUSTOMER.   <4>
   MOVE CUS-HISTORY TO LIST-PURCHASE.
   SET IND-DET TO 1.
   MOVE 1 TO END-FILE.
   PERFORM READ-DETAIL
      UNTIL END-FILE = 0
      OR IND-DET = 21.
   MOVE LIST-DETAIL TO ORD-DETAIL.  <2>
   WRITE COM
      INVALID KEY DISPLAY "ERROR".
   MOVE LIST-PURCHASE
      TO CUS-HISTORY.               <3>
   REWRITE CLI
      INVALID KEY DISPLAY "ERROR CUS".

READ-CUS-CODE.
   DISPLAY "CUSTOMER NUMBER: "
      WITH NO ADVANCING.
   ACCEPT CUS-CODE.
   MOVE 0 TO END-FILE.
   READ CUSTOMER INVALID KEY
      DISPLAY "NO SUCH CUSTOMER"
      MOVE 1 TO END-FILE
   END-READ.
```



```
READ-DETAIL.
   DISPLAY "PRODUCT CODE (0 = END): ".
   ACCEPT PROD-CODE.
   IF PROD-CODE = "0"
      MOVE 0
         TO REF-DET-STK(IND-DET)<12>
      MOVE 0 TO END-FILE
   ELSE
      PERFORM READ-PROD-CODE.

READ-PROD-CODE.
   MOVE 1 TO EXIST-PROD.
   MOVE PROD-CODE TO STK-CODE.      <5>
   READ STOCK INVALID KEY
   MOVE 0 TO EXIST-PROD.
   IF EXIST-PROD = 0
      DISPLAY "NO SUCH PRODUCT"
   ELSE
      PERFORM UPDATE-ORD-DETAIL.

UPDATE-ORD-DETAIL.
   MOVE 1 TO NEXT-DET.
   DISPLAY "QUANTITY ORDERED: "
      WITH NO ADVANCING
   ACCEPT ORD-QTY(IND-DET).
   PERFORM UNTIL
      REF-DET-STK(NEXT-DET)
         = PROD-CODE<9>
      OR IND-DET = NEXT-DET
      ADD 1 TO NEXT-DET
   END-PERFORM.
   IF IND-DET = NEXT-DET          <10>
      MOVE PROD-CODE
        TO REF-DET-STK(IND-DET)    <6>
      PERFORM UPDATE-CUS-HISTO
      SET IND-DET UP BY 1
   ELSE
     DISPLAY "ERROR: ALREADY ORDERED".

UPDATE-CUS-HISTO.
   SET IND TO 1.
   PERFORM UNTIL
      REF-PURCH-STK(IND) = PROD-CODE
         OR REF-PURCH-STK(IND) = 0
        OR IND = 101               <7>
      SET IND UP BY 1
   END-PERFORM.
   IF IND = 101
      DISPLAY "ERR: HISTORY OVERFLOW"
      EXIT.
      IF REF-PURCH-STK(IND)
        = PROD-CODE              <11>
         ADD ORD-QTY(IND-DET)
             TO TOT(IND)
      ELSE
         MOVE PROD-CODE
          TO REF-PURCH-STK(IND)    <8>
         MOVE ORD-QTY(IND-DET)
            TO TOT(IND).

LIST-ORD.
   DISPLAY "LIST OF ORDERS ".
   CLOSE ORDER.
   OPEN I-O ORDER.
   MOVE 1 TO END-FILE.
  PERFORM READ-ORD UNTIL END-FILE = 0.

READ-ORD.
   READ ORDER NEXT
      AT END MOVE 0 TO END-FILE
      NOT AT END
         DISPLAY "ORD-CODE "
            WITH NO ADVANCING
         DISPLAY ORD-CODE
         DISPLAY "ORD-CUSTOMER "
            WITH NO ADVANCING
         DISPLAY ORD-CUSTOMER
         DISPLAY "ORD-DETAIL "
       MOVE ORD-DETAIL TO LIST-DETAIL
         SET IND-DET TO 1
         MOVE 1 TO END-DETAIL
         PERFORM DISPLAY-DETAIL.

INIT-HIST.                       <13>
   SET IND TO 1.
   PERFORM UNTIL IND = 100
      MOVE 0 TO REF-PURCH-STK(IND)
      MOVE 0 TO TOT(IND)
      SET IND UP BY 1
   END-PERFORM.
   MOVE LIST-PURCHASE TO CUS-HISTORY.

DISPLAY-DETAIL.
   IF IND-DET = 21
      MOVE 0 TO END-DETAIL
      EXIT.
   IF REF-DET-STK(IND-DET) = 0
      MOVE 0 TO END-DETAIL
   ELSE
      DISPLAY REF-DET-STK(IND-DET)
      DISPLAY ORD-QTY(IND-DET)
      SET IND-DET UP BY 1.
```

Chapter 10

State of the art and conclusion

16/9/2001

Though reverse engineering data structures still is a complex task, it appears that the current state of the art provides us with sufficiently powerful concepts and techniques to make this enterprise more realistic.

The literature proposes systematic approaches for database schema recovering:

- RE of standard files: [Casanova 1983], [Casanova 1984], [Nillson 1985], [Davis 1985], [Sabanis 1992], [Hainaut 1993b], [Edwards 1995]
- RE of IMS databases: [Navathe 1988], [Winans 1990], [Batini 1992], [Hainaut 1993b], [Fong 1993]
- RE of TOTAL/IMAGE databases: [Hainaut 1993b]
- RE of CODASYL databases: [Batini 1992], [Hainaut 1993b], [Fong 1993], [Edwards 1995]
- RE of relational databases: [Casanova 1984], [Navathe 1988], [Davis 1988], [Johannesson 1990], [Markowitz 1990], [Springsteel 1990], [Kalman 1991], [Fonkam 1992], [Batini 1992], [Premerlani 1993], [Chiang 1993], [Campbell 1994], [Chiang 1994], [Shoval 1993], [Hainaut 1993b], [Petit 1994], [Andersson 1994], [Signore 1994], [Vermeer 1995], [Comyn 1996], [Chiang 1996].
- RE of OO databases: [Hainaut 1997b], [Theodoros 1998].

Most of these studies appear to be limited in scope, and are generally based on assumptions on the quality and completeness of the source data structures to reverse engineer that cannot be relied on in many practical situations. For instance, they often suppose that,

- all the conceptual specifications have been translated into data structures and constraints (at least until 1993),
- the translation is rather straightforward (no tricky representations); for instance, a relational schema often is supposed to be in 4NF; [Premerlani 1993] is one of the first proposals that cope with some non trivial representations;
- the schema has not been deeply restructured for performance objectives or for any other requirements,
- a complete physical schema of the data is available,
- names have been chosen rationally (e.g., a foreign key and the referenced primary key have the same name).

In many proposals, the only databases processable are those that have been obtained by a rigourous database design method. This condition cannot be assumed for most large operational databases, particularly for the oldest ones. Moreover, these proposals are most often dedicated to one data model and do not attempt to elaborate techniques and reasonings common to several models, leaving the question of a general DBRE approach unanswered. Since 1993, some authors recognize that the procedural part of the application programs is an essential source of information on the data structures [Joris 1992], [Hainaut 1993a], [Petit 1994], [Andersson 1994], [Signore 1994].

The list of references mentioned above is far from complete and is likely to include some

near-duplicate papers. Nevertheless, it makes clear that the research emphasis has been put toward reverse engineering relational databases (Figure 10-1). A bit surprisingly, the most important needs expressed by the industry and the administrations concern IMS, COBOL and CODASYL legacy systems.

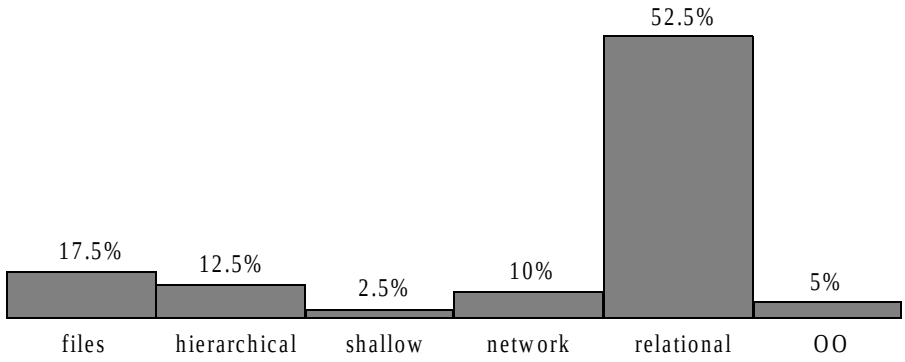


Figure 10-1: Distribution of selected research publications according to the DMS model.

Many reverse engineering problems are now identified. However, solving them in every possible situation is a goal that is far from being reached. In addition, some problems still remain to be worked out. We briefly discuss some of them.

- Though DBRE seems to be sufficiently mature to consider the development of tools and training practitioners, coping with the application as a whole is still an unsolved problem. Indeed, due to the much larger variety of problems and the lower level of formality of practical software engineering, reverse engineering of the procedural components of a large application still is impossible. Moreover, the very meaning of the term *reverse engineering* seems to be different in the database and software comunities. While DBRE aims at recovering the complete abstract specification of an operational component, software reverse engineering seeks for more modest goals forming the *program understanding* domain, still far from recovering the complete abstract specification of the programs. It is clear that DBRE can contribute to program understanding, but the link between them still is to be developed.
- The economics of DBRE is a critical point for which there is no clear proposals so far. In particular, we do not know at the present time how to evaluate the cost of a DBRE project and when to stop a DBRE project, either because we have got enough information or because it proves too costly.
- Reengineering, migrating, reusing and converting system components include the reverse engineering of the existing system. We know how to perform reverse engineering, but there is no clear procedure to carry out the former operations.
- In the early times of DBRE, defining general purpose algorithms and procedures which one could automatically apply was the ultimate goal. It should be clear now that this

goal cannot be achieved but in very simple situations that are seldom observed in the real world. Unfortunately, current CASE tools still are based on this assumption, and offer little more than *SQL code Extractors* and elementary *foreign key to relationship type* converters. According to their users, their main contribution is that of a mere graphical schema editor.

- Many engineering processes that include reverse engineering lead to code production. System conversion, data conversion, system integration and component wrapping are some examples. As shown in Section 2.2, all the activities carried out to build higher level abstractions from the operational code can be completely formalized as specifications transformations. Recording these operations as a history of the DBRE activities provides us with a formal trace of the process. From this trace, one can derive the reverse and forward mappings between the operational code and the conceptual specifications, and, as a consequence, one can automatically generate the various procedures implementing the engineering processes mentioned above. Much work still remains to be done in this direction[1].

About OO reverse engineering

It would be unconceivable to close a presentation on database reverse engineering without a discussion on reverse engineering toward OO specifications. The topic is important, since it makes it possible to include legacy databases into modern distributed object architectures, which is a most promising way of reengineering these valuable assets inherited from the past.

So far, the term OO reverse engineering has been given two distinct interpretations, namely *building an OO description of a standard application* and *building/recovering an OO description of an OO application.* The second kind of problems has been briefly discussed in Section 7.6. We will tell some words on the first one, according to which a standard (typically 3GL) application is analyzed in order to build an OO description of its data objects and of as much as possible of its procedural components. There is a strong trend toward the OO conversion of standard (typically 3GL) applications. Typically, they are analyzed in order to build an OO description of their data objects and of as many as possible parts of their procedural components. A typical overview of a reverse engineering project following this approach consists in finding potential object classes and their basic methods. For example, a COBOL business application based on files CUSTOMER, ITEM and ORDER will be given a description comprising *Customer*, *Item* and *Order* classes, with their associated methods such as *RegisterCustomer*, *DropCustomer*, *ChangeAddress*, *SendInvoice*, etc. The initial idea is quite simple [Sneed 1996]:

- each record type implements an object class and each record field represents a class attribute;
- the creation, destruction and updating methods (e.g. *RegisterCustomer*, *DropCus-*

1. One of the goals of the InterDB project is to develop a DB-MAIN co-processor that automatically generates wrapped data components from DBRE histories [Thiran 1998].

tomer, ChangeAddress) can be discovered by extracted and reordering the procedural sections that manage the source records;

- the application methods (e.g. *SendInvoice*) can be extracted by searching the code for the functional modules.

This idea has been supported by much research effort in the last years [Jacobson 1991] [Gall 1995], [Sneed 1995], [Yeh 1995], [Newcomb 1995]. Unfortunately, it proved much more difficult to implement than originally expected. Indeed, the process of code analysis must take into account complex patterns such as near-duplication (near-identical code sections duplicated throughout the programs [Baker 1995]), interleaving (a single code section used by several execution flows [Rugaber 1995]) and runtime-determined control structures (e.g. dynamically changing the target of a *goto* statement or dynamic SQL). Some authors even propose, in some situations, to leave the code aside, and to reuse the data only [Sneed 1996b], among others through wrapping techniques based on the CORBA model. In addition many extracted modules appear to cope with the management of several record types, i.e. with more than one potential object class. This latter problem forces the analyst to make arbitrary choices, to deeply restructure the code, or to resort to some heuristics [Penteado 1996].
Several code analysis techniques have been proposed to examine the static and dynamic relationships between statements and data structures. Dataflow graphs, dependency graphs and program slicing are among the most popular. In particular, the concept of program slicing seems to be the ultimate solution to locate the potential methods of the record/classes of a program.

Chapter 11

References

24/9/2002

[Aiken 1996] Aiken, P. 1996. *Data Reverse Engineering*, McGraw-Hill.

[Andersson 1994] Andersson, M. 1994. Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag

[Baker 1995] Baker, B. 1995. On Finding Duplication and Near-Duplication in Large Software Systems. Proc. of the 2nd IEEE Working Conf. on Reverse Engineering, Toronto, July 1995, IEEE Computer Society Press.

[Batini 1992] Batini, C., Ceri, S., Navathe, S. 1992. *Conceptual Database Design - An Entity-Relationship Approach*, Benjamin/Cummings

[Batini 1993] Batini, C., Di Battista, G., Santucci, G. 1993. Structuring Primitives for a Dictionary of Entity Relationship Data Schemas, *IEEE TSE*, Vol. 19, No. 4

[Bitton 1989] Bitton, D., Millman, J., Torgersen, S. 1989. A Feasibility and Performance Study of Dependency Inference, in *Proc. IEEE Data Engineering Conference*, Los Angeles.

[Blaha 1995] Blaha, M.R., Premerlani, W., J. 1995. Observed Idiosyncracies of Relational Database designs, in *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press

[Blaha 1996] Blaha, M. 1996. A Catalog of Object Model Transformations, in *Proc. of the 3rd Working Conference on Reverse Engineering* (WCRE'96), Monterey, Nov. 8-10, 1996, IEEE Computer Society Press

[Blaha 1998] Blaha, M., Premerlani, W. 1998. *Object-Oriented Modeling and Design for Database Applications*, Prentice Hall

[Blaha 1998b] Blaha, M., 1998. On Reverse Engineering of Vendor Databases, in *Proc. of the 5th IEEE Working Conf. on Reverse Engineering*, Honolulu, October 1998, IEEE Computer Society Press

[Bolois 1994] Bolois, G., Robillard, P. 1994. Transformations in Reengineering Techniques, in *Proc. of the 4th Reengineering Forum "Reengineering in Practice"*, Victoria, Canada

[Brodie 1995] Brodie, M., Stonebraker, M. 1995. *Migrating Legacy Systems*, Morgan Kaufmann.

[Campbell 1994] Campbell, L., Halpin, T. 1994. The reverse engineering of relational databases, in *Proc. 6th Int. Work. on CASE* (1994).

[Casanova 1983] Casanova, M., Amarel de Sa, J. 1983. Designing Entity Relationship Schemas for Conventional Information Systems, in *Proc. of Entity-Relationship Approach*, pp. 265-278

[Casanova 1984] Casanova, M., A., Amaral De Sa. 1984. Mapping uninterpreted Schemes into Entity-Relationship diagrams: two applications to conceptual schema design, in *IBM J. Res. & Develop.*, Vol. 28, No 1

[Celko 1995] Celko, J. 1995. *SQL for Smarties: Advanced SQL Programming*, Morgan Kaufmann.

[Chiang 1993] Chiang, R., H., Barron, T., M., Storey, V., C. 1993. Performance evaluation of reverse engineering relational databases into Extended ER models, in Proc. of the 12th Conf. on ERA (1993)

[Chiang 1994] Chiang, R., H., Barron, T., M., Storey, V., C. 1994. Reverse Engineering of Relational Databases : Extraction of an EER model from a relational database, *Journ. of Data and Knowledge Engineering*, Vol. 12, No. 2 (March 1994), pp107-142

[Chiang 1996] Chiang, R., H., Barron, T., M., Storey, V., C. 1996. A framework for the design and evaluation of reverse engineering methods for relational databases, *Data and Knowledge Engineering*, Vol. 21, No. 1 (December 1996)

[Chikofski 1990] Chikofski, E., Cross J. II. 1990. Reverse Engineering and design recovery: A taxonomy, IEEE Software, Jan. 1990

[Choobineh 1988] Choobineh, J. et al. 1988. An Expert Database Design System based on Analysis of Forms, *IEEE Trans. on Software Engineering*, 4, 2, Feb. 1988.

[Comyn 1996] Comyn-Wattiau, I., Akoka, J. 1996. Reverse Engineering of Relational Database Physical Schema, in *Proc. 15th Int. Conf. on Conceptual Modeling* (ERA), Cottbus, LNCS 1157, Springer Verlag

[D'Atri 1984] D'Atri, A., Sacca, D. 1984. Equivalence and Mapping of Database Schemes, in *Proc. 10th VLDB conf.*, Singapore

[Davis 1985] Davis, K., H., Arora, A., K. 1985. A Methodology for Translating a Conventional File System into an Entity-Relationship Model, in *Proc. of ERA*, IEEE/ North-Holland

[Davis 1988] Davis, K., H., Arora, A., K. 1988. Converting a Relational Database model to an Entity Relationship Model, in *Proc. of Entity-Relationship Approach: a Bridge to the User*, 1988

[De Troyer 1993] De Troyer, O. 1993. *On data schema transformation*, PhD Thesis, University of Tilburg, Tilburg, The Netherlands

[Edwards 1995] Edwards, H., M., Munro, M. 1995. Deriving a Logical Model for a System Using Recast Method, in *Proc. of the 2nd IEEE WC on Reverse Engineering*, Toronto, IEEE Computer Society Press

[Elmasri 1997] Elmasri, R., Navathe, S. 1997. *Fundamentals of Database Systems*, Benjamin-Cummings

[Fahrner 1995] Fahrner, C., Vossen, G. 1995. A survey of database design transformations based on the Entity-Relationship model, Data Knowledge Eng. 15:3

[Fong 1994] Fong, J., Ho, M. 1994. Knowledge-based Approach for Abstracting Hierarchical and Network Schema Semantics, in *Proc. of the 12th Int. Conf. on ER Approach*, Arlington-Dallas, Springer-Verlag

[Fonkam 1992] Fonkam, M., M., Gray, W., A. 1992. An approach to Eliciting the Semantics of Relational Databases, in *Proc. of 4th Int. Conf. on Advance Information Systems Engineering - CAiSE'92*, pp. 463-480, Springer-Verlag, 1992

[Gall 1995] Gall, H., Klosch, R. 1995. Finding Objects in Procedural Programs: An Alternative Approach, in *Proc. of the 2nd Working Conference on Reverse Engineering* (WCRE'95), Toronto, July 14-16, 1995, IEEE Computer Society Press

[Geller 1989] Geller, J., R. 1989. *IMS, Administration, Programming and Data Base Design*, Wiley

[Hall 1992] Hall, P., A. (Ed.). 1992. *Software Reuse and Reverse Engineering* in Practice, Chapman&Hall.

[Hainaut 1981] Hainaut, J-L. 1981. Theoretical and practical tools for data base design, in *Proc. of the Very Large Databases Conf.*, pp. 216-224, September, 1981

[Hainaut 1989] Hainaut, J.-L. 1989. A Generic Entity-Relationship Model, in *Proc. of the IFIP WG 8.1 Conf. on Information System Concepts: an in-depth analysis*, North-Holland

[Hainaut 1992] Hainaut, J-L., Cadelli, M., Decuyper, B., Marchand, O. 1992. Database CASE Tool Architecture: Principles for Flexible Design Strategies, in Proc. of the 4th Int. Conf. on Advanced Information System Engineering (CAiSE-92), Manchester, May 1992, Springer-Verlag, LNCS

[Hainaut 1993a] Hainaut, J-L., Chandelon M., Tonneau C., Joris M. 1993a. Contribution to a Theory of Database Reverse Engineering, in *Proc. of the IEEE Working Conf. on Reverse Engineering*, Baltimore, May 1993

[Hainaut 1993b] Hainaut, J-L, Chandelon M., Tonneau C., Joris M. 1993b. Transformational techniques for database reverse engineering, in *Proc. of the 12th Int. Conf. on ER Approach*, Arlington-Dallas, E/R Institute and Springer-Verlag, LNCS

[Hainaut 1994] Hainaut, J-L, Englebert, V., Henrard, J., Hick J-M., Roland, D. 1994. Evolution of database Applications: the DB-MAIN Approach, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag

[Hainaut 1995a] Hainaut, J-L. 1995a. Requirements for Information System Reverse Engineering Support, in *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press

[Hainaut 1995b] Hainaut, J-L. 1995b. *Database Reverse Engineering - Problems, Methods and Tools*, Tutorial notes, CAiSE•95, Jyväskylä, Finland, May. 1995 (available at jlh@info.fundp.ac.be)

[Hainaut 1996] Hainaut, J.-L., Hick, J.-M., Englebert, V., Henrard, J., Roland, D. 1996. Understanding implementations of IS-A Relations, in *Proc. of the conference on the ER Approach*, Cottbus, Oct. 1996, LNCS, Springer-Verlag

[Hainaut 1996a] Hainaut, J-L, Henrard, J., Roland, D., Englebert, V., Hick J-M. 1996. Structure Elicitation in Database Reverse Engineering, *Proc. of the IEEE Working Conf. on Reverse Engineering*, Monterey, Nov. 1996, IEEE Computer Society Press

[Hainaut 1996b] Hainaut, J-L, Roland, D., Hick J-M., Henrard, J., Englebert, V. 1996. Database Reverse Engineering: from Requirements to CARE tools, *Journal of Automated Software Engineering*, Vol. 3, No. 1 (1996).

[Hainaut 1996c] Hainaut, J-L, Roland, D., Hick J-M., Henrard, J., Englebert, V. 1996b. Database design recovery, in *Proc. of CAiSE•96*, Springer-Verlag, 1996

[Hainaut 1997] Hainaut, J-L. Hick, J-M., Henrard, J., Roland, D. and Englebert, V. 1997. Knowledge Transfer in Database Reverse Engineering - A supporting Case Study, in *Proc. of the 4th IEEE Working Conf. on Reverse Engineering*, Amsterdam, October 1997, IEEE Computer Society Press

[Hainaut 1997b Hainaut, J-L., Henrard, J., Hick, J-M., Roland, D., Englebert, V. 1997. Contribution to the Reverse Engineering of OO Applications - Methodology and Case Study, in *Proc. of the IFIP 2.6 WC on Database Semantics* (DS-7), Leysin (CH), Oct. 1997, Chapman-Hall

[Halpin 1995] Halpin, T., A., Proper, H., A. 1995. Database schema transformation and optimization, *Proc. of the 14th Int. Conf. on ER/OO Modelling (ERA)*

[Henrard 1998] Henrard, J., Roland, D., Englebert, V., Hick, J-M., Hainaut, J-L. 1998. Program Understanding in Databases Reverse Engineering, in *Proc. of the 9th Int. Conf. on Databases and Expert Systems Applications* (DEXA'98), Vienna, June 1998, Springer-Verlag

[IEEE 1990] *Special issue on Reverse Engineering*, IEEE Software, January, 1990

[Jacobson 1991] Jacobson, I., Lindström, F. 1991 Re-engineering of old systems to an object-oriented architecture, in *Proc of OOPSLA'91*, pp.340-350

[Jajodia 1983] Jajodia, S., Ng, P., A., Springsteel, F., N. 1983. The problem of Equivalence for Entity-Relationship Diagrams, *IEEE Trans. on Soft. Eng.*, SE-9, 5

[Johannesson 1990] Johannesson, P., Kalman, K. 1990. A Method for Translating Relational Schemas into Conceptual Schemas, in *Proc. of the 8th ERA conference*, Toronto, North-Holland,

[Joris 1992] Joris, M., Van Hoe, R., Hainaut, J-L., Chandelon M., Tonneau C., Bodart F. et al. 1992. PHENIX: methods and tools for database reverse engineering, in *Proc. 5th Int. Conf. on Software Engineering and Applications*, Toulouse, December 1992, EC2 Publish.

[Kalman 1991] Kalman, K. 1991. Implementation and critique of an algorithm which maps a relational database to a conceptual model, in Proc. of the CAiSE•91 conference.

[Kobayashi 1986]Kobayashi, I. 1986. Losslessness and Semantic Correctness of Database Schema Transformation: another look of Schema Equivalence, *Information Systems*, Vol. 11, No 1, pp. 41-59

[Lien 1982] Lien, Y., E. 1982. On the equivalence of database models, JACM, 29, 2

[Markowitz 1990]Markowitz, K., M., Makowsky, J., A. 1990. Identifying Extended Entity-Relationship Object Structures in Relational Schemas, *IEEE Trans. on Software Engineering*, Vol. 16, No. 8

[Metaxides 1975] Metaxides, A. 1975. "Information bearing" and "non-information bearing" sets, in *Data Base Description*, Proc. of the IFIP TC2 Work. Conf., Douqué & Nijssen (Ed.), North-Holland

[Mfourga 1997] Mfourga, N. 1997. Extracting Entity-Relationship Schemas from Relational Databases: A Form-Driven Approach, in *Proc. of the 4th Working Conference on Reverse Engineering* (WCRE'97), Amsterdam, Oct. 1997, IEEE Computer Society Press

[Navathe 1980]Navathe, S., B. 1980. Schema Analysis for Database Restructuring, *ACM TODS*, Vol.5, No.2

[Navathe 1988] Navathe, S., B., Awong, A. 1988. Abstracting Relational and Hierarchical Data with a Semantic Data Model, in *Proc. of Entity-Relationship Approach: a Bridge to the User*

[Newcomb 1995]Newcomb P. et al. 1995. Reengineering Procedural Into Object-Oriented Systems, in *Proc. of the 2nd Working Conference on Reverse Engineering* (WCRE'95), Toronto, July 14-16, 1995, IEEE Computer Society Press

[Nilsson 1985] Nilsson,E., G. 1985. The Translation of COBOL Data Structure to an Entity-Rel-type Conceptual Schema, in *Proc. of ERA Conference*, IEEE/North-Holland,

[Penteado 1996]Penteado, R.D., Germano, F.S.R. and Masiero, P.C. 1996. An Overall Process Based on Fusion to Reverse Engineering Legacy Code, in *Proc. of the 3rd IEEE Working Conf. on Reverse Engineering*, Monterey, Nov. 1996, IEEE Computer Society Press.

[Petit 1994] Petit, J-M., Kouloumdjian, J., Bouliaut, J-F., Toumani, F. 1994. Using Queries to Improve Database Reverse Engineering, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag

[Premerlani 1993] Premerlani, W., J., Blaha, M.R. 1993. An Approach for Reverse Engineering of Relational Databases, in *Proc. of the IEEE Working Conf. on Reverse Engineering*, IEEE Computer Society Press

[Rauh 1995] Rauh, O., Stickel, E. 1995. Standard Transformations for the Normalization of ER Schemata, *Proc. of the CAiSE•95 Conf.*, Jyväskylä, Finland, LNCS, Springer-Verlag

[Rock 1990] Rock-Evans, R. 1990. *Reverse Engineering: Markets, Methods and Tools*, OVUM report

[Rosenthal 1988]Rosenthal, A., Reiner, D. 1988. Theoretically sound transformations for Practical Database Design, in *Proc. of the 6th Int. Conf. on Entity-Relationship Approach*, March (Ed.), North-Holland

[Rosenthal 1994]Rosenthal, A., Reiner, D. 1994. Tools and Transformations - Rigorous and Otherwise - for Practical Database Design, *ACM TODS*, Vol. 19, No. 2

[Rugaber 1995]Rugaber, S., Stirewalt, K. and Wills, L. 1995. The Interleaving Problem in Program Understanding. in *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July. 1995, IEEE Computer Society Press.

[Sabanis 1992] Sabanis, N., Stevenson, N. 1992. Tools and Techniques for Data Remodelling Cobol Applications, in *Proc. 5th Int. Conf. on Software Engineering and Applications*, Toulouse, 7-11 December, pp. 517-529, EC2 Publish.

[Selfridge 1993] Selfridge, P., G., Waters, R., C., Chikofsky, E., J. 1993. Challenges to the Field of Reverse Engineering, in *Proc. of the 1st WC on Reverse Engineering*, pp.144-150, IEEE Computer Society Press

[Shoval 1993] Shoval, P., Shreiber, N. 1993. Database Reverse Engineering : from Relational to the Binary Relationship Model, *Data and Knowledge Engineering*, Vol. 10, No. 10

[Signore 1994] Signore, O, Loffredo, M., Gregori, M., Cima, M. 1994. Reconstruction of ER Schema from Database Applications: a Cognitive Approach, in *Proc. of the 13th Int. Conf. on ER Approach, Manchester*, Springer-Verlag

[Signore 1994b] Signore, O, Loffredo, M., Gregori, M., Cima, M. 1994. Using procedural patterns in abstracting relational schemata, in *Proc. IEEE 3rd Workshop on Program Comprehension*

[Sneed 1995] Sneed, H.M. and Nyáry. 1995. Extracting Object-Oriented Specification from Procedurally Oriented Programs. in *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July. 1995, IEEE Computer Society Press.

[Sneed 1996] Sneed, H.S. 1996. Object-Oriented COBOL Recycling. Proc. of the 3rd IEEE Working Conf. on Reverse Engineering, Monterey, Nov. 1996, IEEE Computer Society Press.

[Sneed 1996b] Sneed, H.S. 1996b. Encapsulating Legacy Software for Use in Client/Server Systems. in *Proc. of the 3rd IEEE Working Conf. on Reverse Engineering*, Monterey, Nov. 1996, IEEE Computer Society Press.

[Springsteel 1990] Springsteel, F., N., Kou, C. 1990. Reverse Data Engineering of E-R designed Relational schemas, in *Proc. of Databases, Parallel Architectures and their Applications*, March, 1990

[Teorey 1994] Teorey, T. J. 1994. *Database Modeling and Design: the Fundamental Principles*, Morgan Kaufman

[Theodoros 1998] Theodoros, L., Edwards, H., Bryant, A. 1998. ROMEO: Reverse Engineering from OO Source Code to OMT Design, in *Proc. of the 5th IEEE Working Conf. on Reverse Engineering*, Honolulu, October 1998, IEEE Computer Society Press

[Thiran 1998] Thiran, Ph., Hainaut, J-L., Bodart, S., Deflorenne, A. 1998. Interoperation of Independent, Heterogeneous and Distributed Databases. Methodology and CASE support: the InterDB Approach. in *Proc. of the Int. Conference on Cooperative Information Systems* (CoopIS-98), New-York, August 1998, IEEE Computer Society Press

[Tsichritzis 1977] Tsichritzis, D., C., Lochovsky, F., H. 1977. *Data Base Management Systems*, Academic Press

[Vermeer 1995]Vermeer, M., Apers, P. 1995. Reverse Engineering of Relational Databases, in *Proc. of the 14th Int. Conf. on ER/OO Modelling* (ERA)

[Weiser 1984] Weiser, M. 1984. Program Slicing, *IEEE TSE*, Vol. 10, pp 352-357

[Wills 1995] Wills, L., Newcomb, P., Chikofsky, E., (Eds) 1995. *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press

[Winans 1990] Winans, J., Davis, K., H. 1990. Software Reverse Engineering from a Currently Existing IMS Database to an Entity-Relationship Model, in *Proc. of Entity-Relationship Approach*, pp. 345-360, Oct., 1990

[Yeh 1995] Yeh, A. S., Harris, D. R., Reubenstein, H. B. 1995. Recovering Abstract Data Types and Object Instances from a Conventional Procedural Language, in *Proc. of the 2nd Working Conference on Reverse Engineering* (WCRE'95), Toronto, July 14-16, 1995, IEEE Computer Society Press